11 Principles of Feature Learning

11.1 Introduction

In Chapter 10 we saw how linear supervised and unsupervised learners alike can be extended to perform nonlinear learning via the use of nonlinear functions (or feature transformations) that we engineered ourselves by visually examining data. For example, we expressed a general nonlinear model for regression as a weighted sum of B nonlinear functions of our input as

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 + \cdots + f_B\left(\mathbf{x}\right) w_B \tag{11.1}$$

where f_1 through f_B are nonlinear parameterized or unparameterized functions (or features) of the data, and w_0 through w_B (along with any additional weights internal to the nonlinear functions) are represented in the weight set Θ.

In this chapter we detail the fundamental tools and principles of *feature learning* (or automatic feature engineering) that allow us to automate this task and *learn* proper features from the data itself, instead of *engineering* them ourselves. In particular we discuss how to choose the form of the nonlinear transformations f_1 through f_B, the number B of them employed, as well as how the parameters in Θ are tuned, *automatically* and for *any dataset*.

11.1.1 The limits of nonlinear feature engineering

As we have described in previous chapters, *features* are those defining characteristics of a given dataset that allow for optimal learning. In Chapter 10 we saw how the quality of the mathematical features we can design ourselves is fundamentally dependent on our level of knowledge regarding the phenomenon we were studying. The more we understand (both intellectually and intuitively) about the process generating the data we have at our fingertips, the better we can design features ourselves. At one extreme where we have near perfect understanding of the process generating our data, this knowledge having come from considerable intuitive, experimental, and mathematical reflection, the features we design allow near perfect performance. However, more often than not we know only a few facts, or perhaps none at all, about the data we are analyzing. The universe is an enormous and complicated place, and we have a solid understanding only of how a sliver of it all works.

Most (particularly modern) machine learning datasets have far more than two inputs, rendering visualization useless as a tool for feature engineering. But even in rare cases where data visualization is possible, we cannot simply rely on our own pattern recognition skills. Take the two toy datasets illustrated in Figure 11.1, for example. The dataset on the left is a regression dataset with one-dimensional input and the one on the right is a two-class classification dataset with two-dimensional input. The true underlying nonlinear model used to generate the data in each case is shown by the dashed black lines. We humans are typically taught only how to recognize the simplest of nonlinear patterns *by eye*, including those created by elementary functions (e.g., polynomials of low degree, exponential functions, sine waves) and simple shapes (e.g., squares, circles, ellipses). Neither of the patterns shown in the figure match such simple nonlinear functionalities. Thus, whether or not a dataset can be visualized, human engineering of proper nonlinear features can be difficult if not outright impossible.

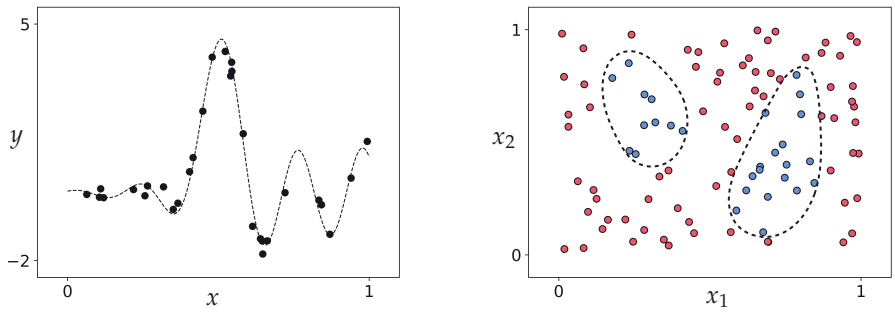

Figure 11.1 Toy (left panel) regression and (right panel) two-class classification datasets that clearly exhibit nonlinear behavior. The true underlying nonlinear function used to generate the data in each case is shown in dashed black. See text for further details.

It is precisely this challenge which motivates the fundamental *feature learning* tools described in this chapter. In short, these technologies *automate* the process of identifying appropriate nonlinear features for arbitrary datasets. With these tools in hand we no longer need to *engineer* proper nonlinearities, at least in terms of how we engineered nonlinear features in the previous chapter. Instead, we aim at *learning* their appropriate forms. Compared to our own limited nonlinear pattern recognition abilities, feature learning tools can identify virtually any nonlinear pattern present in a dataset regardless of its input dimension.

11.1.2 Chapter outline

The aim to automate nonlinear learning is an ambitious one and perhaps at first glance an intimidating one as well, for there are an infinite variety of nonlinearities and nonlinear functions to choose from. How do we, in general,

parse this infinitude automatically to determine the appropriate nonlinearity for a given dataset?

The first step, as we will see in Section 11.2, is to organize the pursuit of automation by first placing the fundamental building blocks of this infinitude into *manageable collections* of (relatively simple) nonlinear functions. These collections are often called *universal approximators*, of which three strains are popularly used and which we introduce here: fixed-shape approximators, artificial neural networks, and trees. After introducing universal approximators we then discuss the fundamental concepts underlying how they are employed, including the necessity for *validation error* as a measurement tool in Section 11.3, a description of *cross-validation* and the *bias-variance trade-off* in Section 11.4, the automatic tuning of nonlinear complexity via *boosting* and *regularization* in Sections 11.5 and 11.6, respectively, as well as the notion of *testing error* in Section 11.7 and *bagging* in Section 11.9.

11.1.3 The complexity dial metaphor of feature learning

The ultimate aim of feature learning is a paradigm for the appropriate and automatic learning of features for any *any dataset* regardless of problem type. This translates – formally speaking – into the the automatic determination of both the proper *form* of the general nonlinear model in Equation (11.1) and the proper *parameter tuning* of this model regardless of training data and problem type. We can think about this challenge metaphorically as (i) the *construction* of, and (ii) the *automatic setting* of, a "complexity dial," like the one illustrated in Figure 11.2 for a simple nonlinear regression dataset (first used in Example 10.1). This complexity dial conceptualization of feature learning visually depicts the challenge of feature learning at a high level as a dial that must be built and automatically tuned to determine the appropriate amount of model complexity needed to represent the phenomenon generating a given dataset.

Setting this complexity dial all the way to the left corresponds, generally speaking, to choosing a model with lowest nonlinear complexity (i.e., a linear model, as depicted visually in the figure). As the dial is turned from left to right various models of increasing complexity are tried against the training data. If turned too far to the right the resulting model will be too complex (or too "wiggly") with respect to the training data (as depicted visually in the two small panels on the right side of the dial). When set "just right" (as depicted visually in the small image atop the complexity dial that is second to the left) the resulting model represents the data – as well as the underlying phenomenon generating it – very well.

While the complexity dial is a simplified depiction of feature learning we will see that it is nonetheless a helpful metaphor, as it will help us organize our understanding of the diverse set of ideas involved in performing it properly.

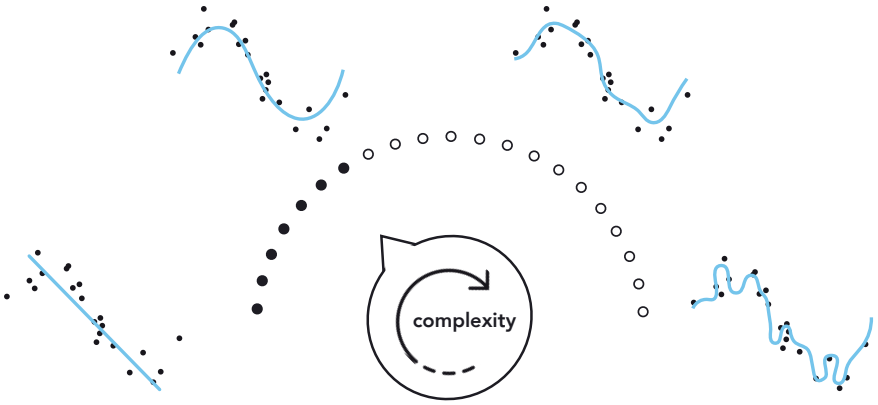

Figure 11.2 A visual depiction of feature learning as the *construction* and *automatic setting* of a "complexity dial" that – broadly speaking – controls the form the nonlinear model in Equation (11.1) as well as its parameter tuning, and thus the complexity of the model with respect to the training data.

11.2 Universal Approximators

In the previous chapter we described how to engineer appropriate nonlinear features ourselves to match the patterns we gleamed in simple datasets. However, very rarely in practice can we design perfect or even strongly-performing nonlinear features by completely relying on our own understanding of a dataset, whether this is gained by visualizing the data, philosophical reflection, or domain expertise.

In this section we jettison the unrealistic assumption that proper nonlinear features can be engineered in the manner described in the previous chapter, and replace it with an equally unrealistic assumption that has far more practical repercussions (as we will see in the forthcoming sections): that we have *complete* and *noiseless* access to the phenomenon generating our data. Here we will see, in the case where we have such unfettered access to data, that absolutely perfect features can be *learned* automatically by combining elements from a set of basic feature transformations, known as *universal approximators*. In this section we will also see elementary exemplars from the three most popular universal approximators, namely, *fixed-shape* approximators, *neural networks*, and *trees*.

For the sake of simplicity we will restrict our discussion to nonlinear regression and two-class classification, which as we saw in Chapter 10, share the same generic nonlinear model, formed as a linear combination of B nonlinear feature transformations of the input

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 + \cdots + f_B\left(\mathbf{x}\right) w_B. \tag{11.2}$$

Recall that with nonlinear two-class classification, we simply pass the nonlinear regression model in Equation (11.2) through the mathematical sign function to make binary predictions. While our focus in this section will be on these two supervised problems, because the general nonlinear model in Equation (11.2) is used in virtually all other forms of nonlinear learning including multi-class classification (see Section 10.4) and unsupervised learning (see Section 10.6), the thrust of the story unveiled here holds more generally for all machine learning problems.

11.2.1 Perfect data

We now start by imagining the impossible: a *perfect* dataset for regression. Such a dataset has two important characteristics: it is *completely noiseless* and *infinitely large*. Being completely noiseless, the first characteristic means that we could completely trust the quality of every one of its input/output pairs. Being infinitely large, means that we have unfettered access to every input/output pair $\left(\mathbf{x}_p, y_p\right)$ of the dataset that could possibly exist. Combined, such a dataset *perfectly* describes the phenomenon that generates it. In the top panels of Figure 11.3 we illustrate what such a perfect dataset would look like in the simplest instance where the input/output data is related linearly.

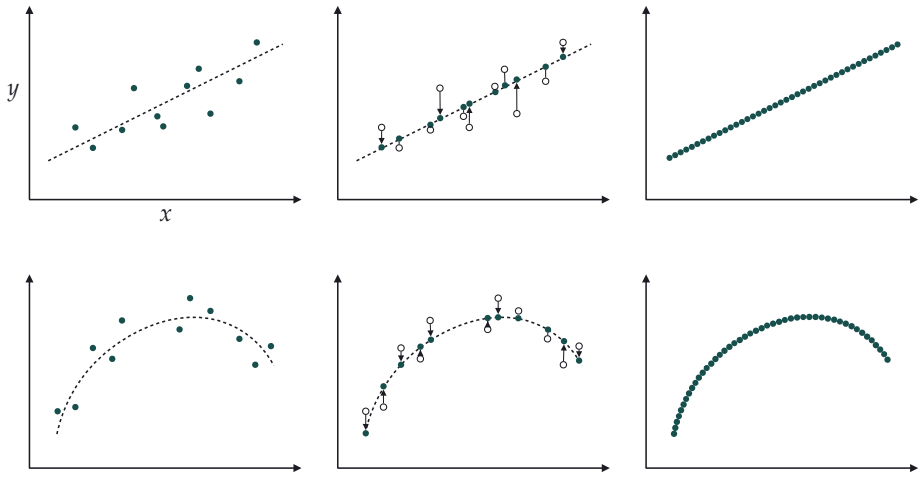

Figure 11.3 (top-left panel) A prototypical realistic linear regression dataset is a noisy and (relatively) small set of points that can be roughly modeled by a line. (top-middle panel) The same dataset with all noise removed from each output. (top-right panel) The perfect linear regression dataset where we have infinitely many points lying precisely on a line. (bottom-left panel) A prototypical realistic nonlinear regression dataset is a noisy and (relatively) small set of points that can be roughly modeled by a nonlinear curve. (bottom-middle panel) All noise removed from the output, creating a noise-free dataset. (bottom-right panel) The perfect nonlinear regression dataset where we have infinitely many points lying precisely on a curve.

Starting in the left panel we show a *realistic* dataset (the kind we deal with in practice) that is both *noisy* and *small*. In the middle panel we show the same dataset, but with the noise removed from each output. In the right panel we depict a *perfect* dataset by adding all missing points from the line to the noiseless data in the middle panel, making the data appear as a continuous line (or hyperplane, in higher dimensions). In the bottom panels of Figure 11.3 we show a similar transition for a prototypical nonlinear regression dataset wherein the perfect data (shown in the rightmost panel) carves out a *continuous* nonlinear curve (or surface, in higher dimensions).

With two-class classification a perfect dataset (using label values $y_p \in \{-1, +1\}$ by default) would share the same characteristics: it is *completely noiseless* and *infinitely large*. However, in this case, the perfect data would appear not as continuous curve or surface itself, but a step function with a *continuous* nonlinear boundary between its top and bottom steps. This is illustrated in Figure 11.4, which mirrors very closely what we saw with regression in Figure 11.3.

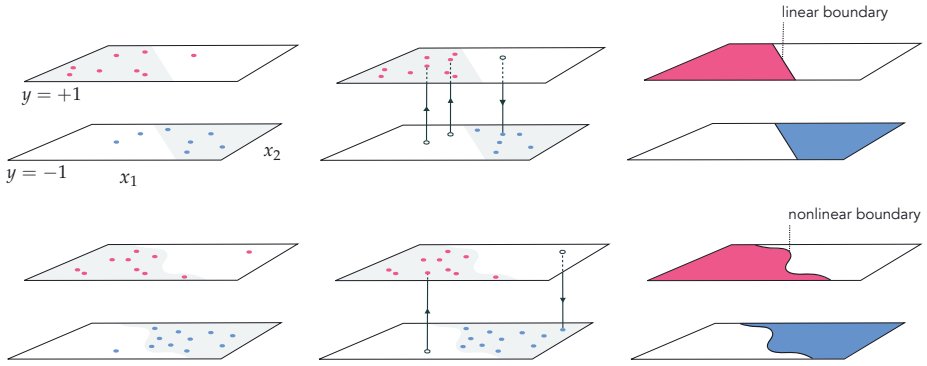

Figure 11.4 (top-left panel) A prototypical realistic linear two-class classification dataset is a noisy and (relatively) small set of points that can be roughly modeled by a step function with linear boundary. (top-middle panel) We progress to remove all noise from the data by returning the true label values to our noisy points. (top-right panel) The perfect linear two-class classification dataset where we have infinitely many points lying precisely on a step function with linear boundary. (bottom-left panel) A prototypical realistic nonlinear two-class classification dataset is a noisy and (relatively) small set of points that can be roughly modeled by a step function with nonlinear boundary. (bottom-middle panel) We progress to remove all noise from the data, creating a noise-free dataset. (bottom-right panel) The perfect nonlinear two-class classification dataset where we have infinitely many points lying precisely on a step function with nonlinear boundary.

In short, a perfect regression dataset is a continuous function with unknown equation. Because of this we will refer to our perfect data using the function notation $y(\mathbf{x})$, meaning that the data pair defined at input $\mathbf{x}$ can be written as either $(\mathbf{x}, y(\mathbf{x}))$ or likewise $(\mathbf{x}, y)$. In the same vein a perfect two-class classification dataset can be represented as a step function $\text{sign}(y(\mathbf{x}))$ with a continuous

boundary – determined by $y(\mathbf{x})$. It is important to bear in mind that the function notation $y(\mathbf{x})$ does not imply that we have knowledge of a closed-form *formula* relating the input/output pairs of a perfect dataset; we do not! Indeed our aim next is to understand how such a formula can be devised to adequately represent a perfect dataset.

11.2.2 The spanning set analogy for universal approximation

Here we will leverage our knowledge and intuition about basic linear algebra concepts such as vectors, spanning sets, and the like (see Section 8.2) to better understand how we can combine nonlinear functions to model perfect regression and classification data. In particular, we will see how vectors and nonlinear functions are very much akin when it comes to the notions of linear combination and spanning sets.

Linear combinations of vectors and functions

To begin, assume we have a set of B vectors $\{\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_B\}$, each having length N. We call this a *spanning* set of vectors. Then, given a particular set of weights w_1 through w_B, the linear combination

$$\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + \cdots + \mathbf{f}_B w_B = \mathbf{y} \tag{11.3}$$

defines a new N-dimensional vector $\mathbf{y}$. This is illustrated in the top row of Figure 11.5 for a particular set of vectors and weights where $B = 3$ and $N = 3$.

The arithmetic of nonlinear functions works in an entirely similar manner: given a spanning set of B nonlinear functions $\{f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_B(\mathbf{x})\}$ (where the input $\mathbf{x}$ is N-dimensional and output is scalar), and a corresponding set of weights, the linear combination

$$w_0 + f_1(\mathbf{x})\, w_1 + f_2(\mathbf{x})\, w_2 + \cdots + f_B(\mathbf{x})\, w_B = y(\mathbf{x}) \tag{11.4}$$

defines a new function $y(\mathbf{x})$. This is illustrated in the bottom row of Figure 11.5 for a particular set of functions and weights where $B = 3$ and $N = 1$.

Notice the similarity between the vector and function arithmetic in Equations (11.3) and (11.4): taking a particular linear combination of a set of vectors creates a new vector with qualities inherited from each vector $\mathbf{f}_b$ in the set, just as taking a linear combination of a set of functions creates a new function taking on qualities of each function $f_b(\mathbf{x})$ in that set. One difference between the two linear combination formulae is the presence of a *bias parameter* w_0 in Equation (11.4). This bias parameter could be rolled into one of the nonlinear functions and not made explicit (by adding a constant function to the mix), but we choose to leave it out-front of the linear combination of functions (as we did with linear models in previous chapters). The sole purpose of this bias parameter is to move our linear combination of functions vertically along the output axis.

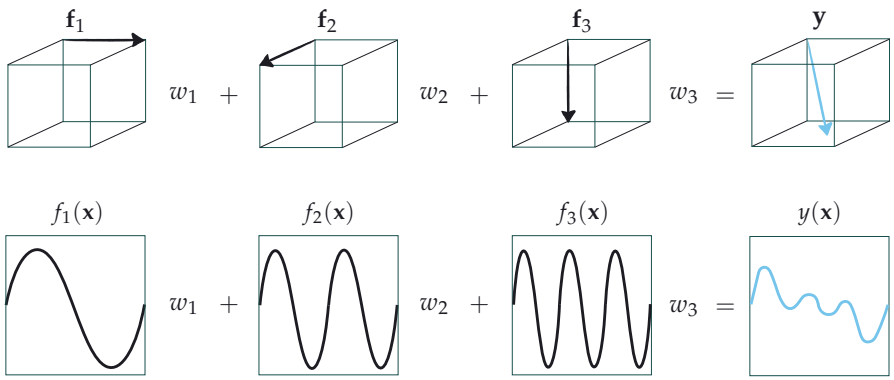

Figure 11.5 (top row) A particular linear combination of vectors $\mathbf{f}_1$, $\mathbf{f}_2$, and $\mathbf{f}_3$ (shown in black) creates a new vector $\mathbf{y}$ (shown in blue). (bottom row) In an entirely similar fashion a particular linear combination of three functions $f_1(x)$, $f_2(x)$, and $f_3(x)$ (shown in black) creates a new function $y(x)$ (shown in blue).

Capacity of spanning sets

Computing the vector $\mathbf{y}$ in Equation (11.3) for a *given* set of weights w_1 through w_B is a trivial affair. The inverse problem on the other hand, i.e., finding the weights given $\mathbf{y}$, is slightly more challenging. Stated algebraically, we want to find the weights w_1 through w_B such that

$$\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + \cdots + \mathbf{f}_B w_B \approx \mathbf{y} \tag{11.5}$$

holds as well as possible. This is illustrated for a simple example in the top row of Figure 11.6.

How well the *vector approximation* in Equation (11.5) holds depends on three crucial and interrelated factors: (i) the diversity (i.e., linear independence) of the spanning vectors, (ii) the number B of them used (in general the larger we make B the better), and (iii) how well we tune the weights w_1 through w_B via minimization of an appropriate cost.[1]

Factors (i) and (ii) determine a spanning set's *rank* or *capacity*, that is a measure for the range of vectors $\mathbf{y}$ we can possibly represent with such a spanning set. A spanning set with a *low* capacity, that is one consisting of a nondiverse and/or a small number of spanning vectors can approximate only a tiny fraction of those present in the entire vector space. On the other hand, a spanning set with a *high* capacity can represent a broader swath of the space. The notion of capacity for

[1] For instance, here we can use the Least Squares cost

$$g(w_1, w_2, ..., w_B) = \left\| \mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + \cdots + \mathbf{f}_B w_B - \mathbf{y} \right\|_2^2. \tag{11.6}$$

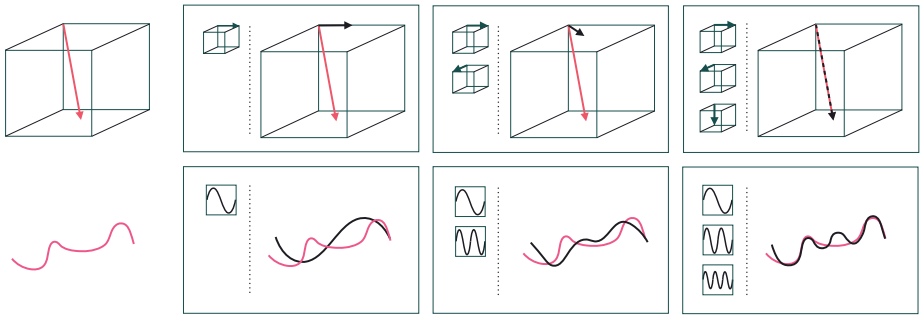

Figure 11.6 (top panels) A three-dimensional vector $\mathbf{y}$ (shown in red in the first panel from the left) is approximated using one (second panel from the left), two (third panel from the left), and three (fourth panel from the left) spanning vectors (here, standard basis vectors). As the number of spanning vectors increases we can approximate $\mathbf{y}$ with greater precision. (bottom panels) The same concept holds with functions as well. A continuous function with scalar input $y(x)$ (shown in red) is approximated using one (second panel from the left), two (third panel from the left), and three (fourth panel from the left) spanning functions (here, sine waves of varying frequency). As the number of functions increases we can approximate $y(x)$ with greater precision.

a spanning set of vectors is illustrated for a particular spanning set in the top row of Figure 11.7.

Turning our attention from vectors to functions, notice that computing the function $y(\mathbf{x})$ in Equation (11.4) for a *given* set of weights w_1 through w_B is straightforward. As with the vector case, we can reverse this problem and try to find the weights w_1 through w_B, for a given $y(\mathbf{x})$, such that

$$w_0 + f_1(\mathbf{x})\,w_1 + f_2(\mathbf{x})\,w_2 + \cdots + f_B(\mathbf{x})\,w_B \approx y(\mathbf{x}) \tag{11.7}$$

holds as well as possible. This is illustrated for a simple example in the bottom row of Figure 11.6.

Once again, how well this *function approximation* holds depends on three crucial and interrelated factors: (i) the diversity of the spanning functions, (ii) the number B of them used, and (iii) how well we tune the weights w_0 through w_B (as well as any parameters internal to our nonlinear functions) via minimization of an appropriate cost.[2]

In analogy to the vector case, factors (i) and (ii) determine the *capacity* of a spanning set of functions. A *low* capacity spanning set that uses a nondiverse and/or small array of nonlinear functions is only capable of representing a small

[2] For instance, here we can use the Least Squares cost

$$g(w_0, w_1, ..., w_B) = \int_{\mathbf{x}\in\mathcal{D}} (w_0 + f_1(\mathbf{x})\,w_1 + \cdots + f_B(\mathbf{x})\,w_B - y(\mathbf{x}))^2\,d\mathbf{x} \tag{11.8}$$

where $\mathcal{D}$ is any desired portion of the input domain.

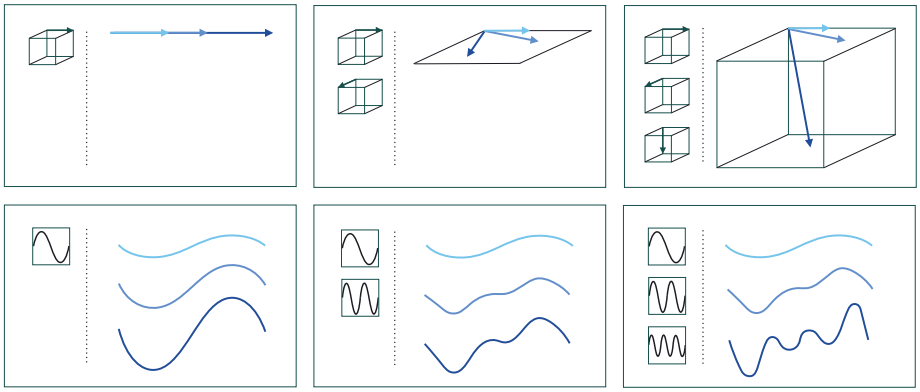

Figure 11.7 (top panels) As we increase the number of (diverse) vectors in a spanning set, from one in the left panel to two and three in the middle and right panels, respectively, we increase the capacity of the spanning set. This is reflected in the increasing diversity of sample vectors created using each spanning set (shown in different shades of blue). (bottom panels) The same concept holds with functions as well. As we increase the number of (diverse) functions in the spanning set, from a single function in the left panel to two and three functions in the middle and right panels, respectively, we increase the spanning set's capacity. This is reflected in the increasing diversity of sample functions created using each spanning set (shown in different shades of blue).

range of nonlinear functions. On the other hand, a spanning set with a *high* capacity can represent a wider swath of functions. The notion of capacity for a spanning set of functions is illustrated for a particular spanning set in the bottom row of Figure 11.7.

Sometimes the spanning functions f_1 through f_B are parameterized, meaning that they have internal parameters themselves. An unparameterized spanning function is very much akin to a spanning vector, as they are both parameter-free. A parameterized spanning function on the other hand can take on a variety of shapes alone, and thus can itself have high capacity. The same cannot be said about spanning vectors and unparameterized spanning functions. This concept is illustrated in Figure 11.8 where in the left column we show an ordinary spanning vector $\mathbf{x} = [1 \ 1]^T$ (top-left panel) along with an unparameterized spanning function, i.e., $\sin(x)$ (bottom-left panel). In the bottom-right panel of the figure we show the parameterized function $\sin(wx)$, which can represent a wider range of different functions as its internal parameter w is adjusted. Thinking analogously, we can also parameterize the spanning vector $\mathbf{x}$, e.g., via multiplying it by the rotation matrix

$$R_w = \begin{bmatrix} \cos(w) & -\sin(w) \\ \sin(w) & \cos(w) \end{bmatrix} \tag{11.9}$$

that allows it to rotate in the plane and represent a range of different vectors depending on how the rotation angle w is set.

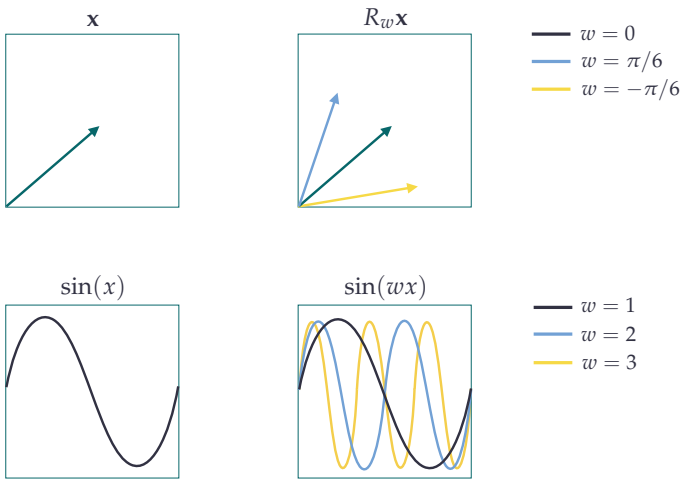

Figure 11.8 (top-left panel) An ordinary spanning vector. (bottom-left panel) An unparameterized spanning function. (bottom-right panel) A parameterized spanning function with a single internal parameter. By changing the value of this internal parameter it can be made to take on a range of shapes. (top-right panel) A parameterized spanning vector (premultiplied by the rotation matrix in Equation (11.9)) changes direction depending on how the parameter w is set.

Universal approximation

In the case of vector-based approximation in Equation (11.5) if we choose $B \geq N$ vectors for our spanning set, and at least N of them are linearly independent, then our spanning set has maximal capacity and we can therefore approximate *every* N-dimensional vector $\mathbf{y}$ to *any* given precision, provided we tune the parameters of the linear combination properly. Such a set of spanning vectors, of which there are infinitely many for an N-dimensional vector space, can approximate (or in this case perfectly represent) every vector *universally*, and is thus sometimes referred to as a *universal approximator*. For example, the simple standard basis (see Exercise 8.1) for a vector space is a common example of a spanning set that is a universal approximator. This notion of universal approximation of vectors is illustrated in the top panel of Figure 11.9.

The same concept holds with function approximation in Equation (11.7) as well. If we choose the right kind of spanning functions, then our spanning set has maximal capacity and we can therefore approximate *every* function $y(\mathbf{x})$ to *any* given precision, provided we tune the parameters of the linear combination properly. Such a set of spanning functions, of which there are infinitely many varieties, can approximate every function *universally*, and is thus often referred

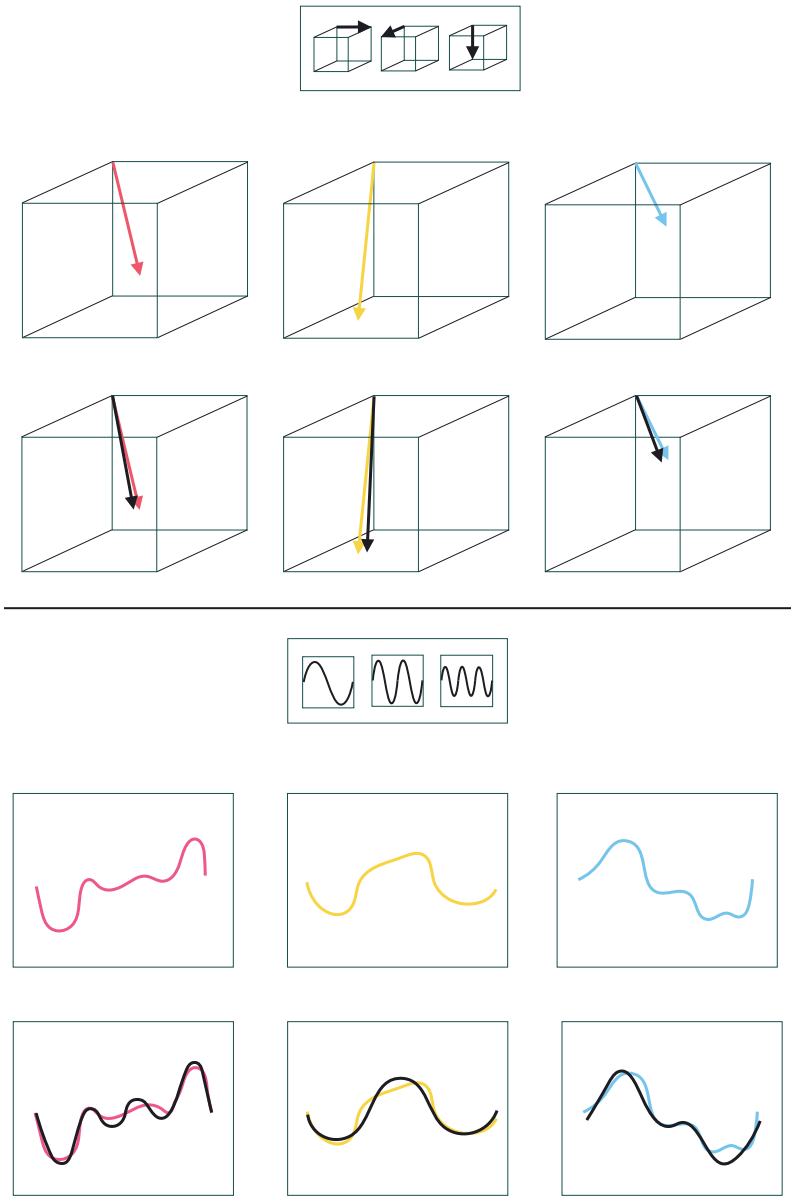

Figure 11.9 (top panel) Universal approximation illustrated in the vector case. (top row) A universal approximator spanning set consisting of three vectors, shown in black. (middle row) Three example vectors to approximate colored red, yellow, and blue, from left to right. (bottom row) The approximation of each vector in the middle row using the spanning set in the top row, shown in black in each instance. This approximation can be made perfect, but for visualization purposes is shown slightly offset here. (bottom panel) The analogous universal approximation scenario illustrated for functions. (top row) A universal approximator spanning set consisting of three functions (in practice many more spanning functions may be needed than shown here). (middle row) Three example functions to approximate colored red, yellow, and blue, from left to right. (bottom row) The approximation of each function in the middle row using the spanning set in the top row, shown in black in each instance.

to as a *universal approximator*. This notion of universal approximation of functions is illustrated in the bottom panel of Figure 11.9.

One difference between the vector and the function regime of universal approximation is that with the latter we may need infinitely many spanning functions to be able to approximate a given function to an arbitrary precision (whereas with the former it is always sufficient to set B greater than or equal to N).

11.2.3 Popular universal approximators

When it comes to approximating functions there is an enormous variety of spanning sets that are *universal approximators*. Indeed, just as in the vector case, with functions there are infinitely many universal approximators. However, for the purposes of organization, convention, as well as a variety of technical matters, universal approximators used in machine learning are often lumped into three main categories referred to as *fixed-shape* approximators, *neural networks*, and *trees*. Here we introduce only the most basic exemplar from each of these three categories, which we will reference throughout the remainder of the chapter. Each of these popular families has its own unique practical strengths and weaknesses as a universal approximator, a wide range of technical details to explore, and conventions of usage.

Example 11.1 The fixed-shape family of universal approximators

The family of *fixed-shape* functions consists of groups of nonlinear functions with no internal parameters, a popular example being *polynomials*.[3] When dealing with just one input this subfamily of fixed-shape functions consists of

$$f_1(x) = x, \;\; f_2(x) = x^2, \;\; f_3(x) = x^3, \;\; \text{etc.}, \tag{11.10}$$

with the Dth element taking the form $f_D(x) = x^D$. A combination of the first D units from this subfamily is often referred to as a *degree-D* polynomial. There are an infinite number of these functions (one for each positive whole number D) and they are *naturally ordered* by their degree. The fact that these functions have no tunable internal parameters gives each a *fixed shape* as shown in the top row of Figure 11.10.

With two inputs x_1 and x_2, a general degree-D polynomial unit takes the analogous form

$$f_b(x_1, x_2) = x_1^p x_2^q \tag{11.11}$$

where p and q are nonnegative integers and $p + q \leq D$. Classically, a degree-D

[3] Polynomials were the first provable universal approximators, this having been shown in 1885 via the so-called (Stone–) Weierstrass approximation theorem (see, e.g., [49]).

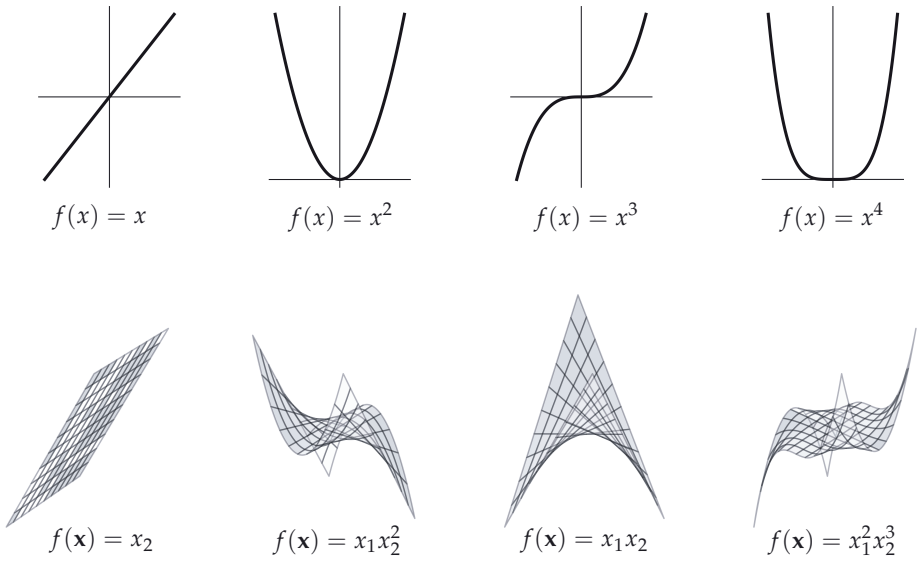

Figure 11.10 Four units from the polynomial family of fixed-shape universal approximators with $N = 1$ (top row) and $N = 2$ (bottom row) dimensional input.

polynomial is a linear combination of all such units. Furthermore, the definition in Equation (11.11) directly generalizes to higher-dimensional input as well. Fixed-shape approximators are discussed in much greater detail in Chapter 12.

Example 11.2 The neural network family of universal approximators

Another popular family of universal approximators are *neural networks*.[4] Broadly speaking neural networks consist of *parameterized* functions,[5] allowing them to take on a variety of different shapes (unlike the fixed-shape functions described previously, each of which takes on a single fixed form).

The simplest subfamily of neural networks consists of parameterized elementary functions (e.g., tanh) of the form

$$f_b(x) = \tanh\left(w_{b,0} + w_{b,1}x\right) \tag{11.12}$$

where the internal parameters $w_{b,0}$ and $w_{b,1}$ of the bth unit allow it to take on a variety of shapes. In the top row of Figure 11.11 we illustrate this fact by randomly setting the values of its two internal parameters, and plotting the result.

To construct neural network features taking in higher-dimensional input we take a linear combination of the input and pass the result through the nonlinear

[4] Neural networks were shown to be universal approximators in the late 1980s and early 1990s [50, 51, 52].

[5] An evolutionary step between fixed-shape and neural network units, that is a network unit whose internal parameters are randomized and fixed, are also universal approximators [53, 54].

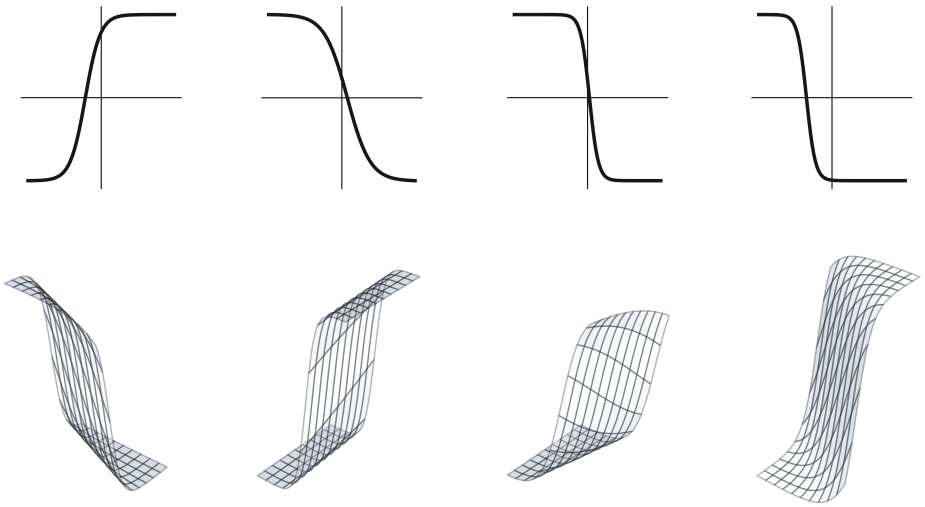

Figure 11.11 Unlike fixed-shape approximators, neural network units are flexible and can take on a variety of shapes based on how we set their internal parameters. Four such units, taking in $N = 1$ (top row) and $N = 2$ (bottom row) dimensional input, are shown whose internal parameters are set randomly in each instance.

function (here, tanh). For example, an element f_b for general N-dimensional input takes the form

$$f_b(\mathbf{x}) = \tanh\left(w_{b,0} + w_{b,1}x_1 + \cdots + w_{b,N}x_N\right). \tag{11.13}$$

As with the lower-dimensional example in Equation (11.12), each function in Equation (11.13) can take on a variety of different shapes, as illustrated in the bottom row of Figure 11.11, based on how we tune its internal parameters. Neural network approximators are described in much greater detail in Chapter 13.

Example 11.3 The trees family of universal approximators

Like neural networks, a single element from the family of tree-based universal approximators[6] can take on a wide array of shapes. The simplest sort of tree unit consists of discrete step functions or, as they are more commonly referred to, *stumps* whose break lies along a single dimension of the input space. A stump with one-dimensional input x can be written as

$$f_b(x) = \begin{cases} v_1 & x \leq s \\ v_2 & x > s \end{cases} \tag{11.14}$$

where s is called a *split point* at which the stump changes values, and v_1 and

[6] Trees have been long known to be universal approximators. See, e.g., [49, 55].

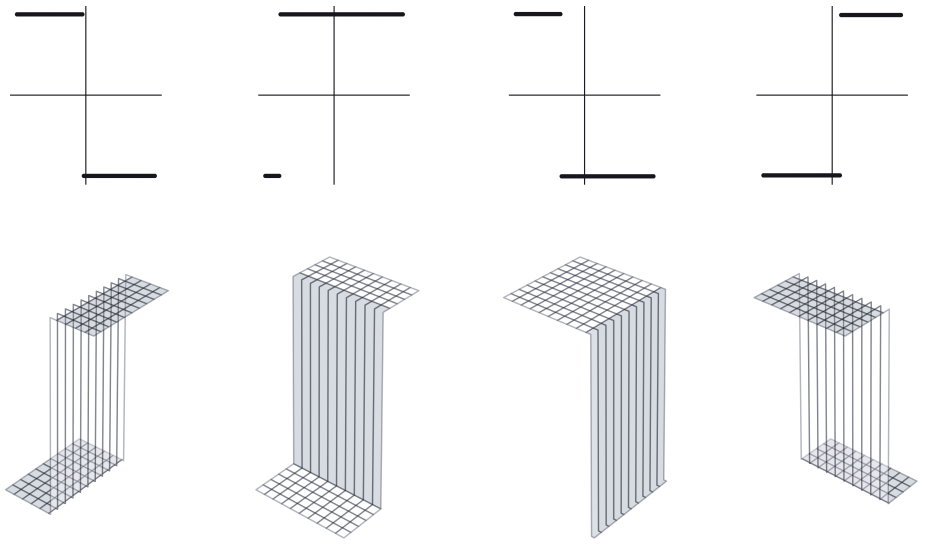

Figure 11.12 Tree-based units can take on a variety of shapes depending on how their split points and leaf values are assigned. Four instances of an $N = 1$ (top row) and $N = 2$ (bottom row) dimensional stump.

v_2 are values taken by the two sides of the stump, respectively, which we refer to as *leaves* of the stump. A tree-based universal approximator is a set of such stumps with each unit having its own unique split point and leaf values.

In the top row of Figure 11.12 we plot four instances of such a stump unit. Higher-dimensional stumps follow this one dimensional pattern. A split point s is first chosen along a *single* input dimension. Each side of the split is then assigned a single leaf value, as illustrated in the bottom row of Figure 11.12 for two-dimensional input. Tree-based approximators are described in much further detail in Chapter 14.

When forming a basic universal approximator based nonlinear model

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 + \cdots + f_B\left(\mathbf{x}\right) w_B \tag{11.15}$$

we always use units from a *single* type of universal approximator (e.g., all fixed-shape, neural network, or tree-based units). In other words, we do not "mix and match," taking a few units from each of the main families. As we will see in the present chapter as well as those following this one, by restricting a model's feature transformations to a single family we can (in each of the three cases) better optimize the learning process and better deal with each family's unique technical eccentricities relating to fundamental scaling issues associated with

fixed-shape units, the nonconvexity of cost functions associated with neural network units, and the discrete nature of tree-based units.

11.2.4 The capacity and optimization dials

With any of the major universal approximators introduced previously (whether they be fixed-shape, neural networks, or trees) we can attain universal approximation to any given precision, provided that the generic nonlinear model in Equation (11.15) has sufficiently large *capacity* (which can be ensured by making B large enough), and that its parameters are tuned sufficiently well through *optimization* of an associated cost function. The notions of *capacity* and *optimization* of such a nonlinear model are depicted conceptually in Figure 11.13 as a set of two *dials*.

The *capacity dial* visually summarizes the amount of capacity we allow into a given model, with each notch on the dial denoting a distinct model constructed from units of a universal approximator. When set all the way to the left we admit as little capacity as possible, i.e., we employ a *linear* model. As we move the capacity dial from left to right (clockwise) we adjust the model, adding more and more capacity, until the dial is set all the way to the right. When set all the way to the right we can imagine admitting an infinite amount of capacity in our model (e.g., by using an infinite number of units from a particular family of universal approximators).

The *optimization dial* visually summarizes how well we minimize the cost function of a given model whose capacity is already set. The setting all the way to the left denotes the initial point of whatever local optimization technique we use. As we turn the optimization dial from left to right (clockwise) we can imagine moving further and further along the particular optimization run we use to properly tune the parameters of the model, with the final step being represented visually as the dial set all the way to the right where we imagine we have successfully minimized the associated cost function.

Note that in this conceptualization each pair of settings (of capacity and optimization dials) produces a unique tuned model: the model's overall architecture/design is decided by the capacity dial, and the set of specific values for the model parameters is determined by the optimization dial. For example, one particular setting may correspond to a model composed of $B = 10$ neural networks units, whose parameters are set by taking 1000 steps of gradient descent, while another setting corresponds to a model composed of $B = 200$ neural networks units, whose parameters are set by taking only five steps of gradient descent.

With these two dial conceptualizations in mind, we can think about the concept of universal approximation of a continuous function as turning *both* dials all the way to the right, as shown in the bottom row of Figure 11.13. That is, to approximate a given continuous function using a universal approximator, we set our model capacity as *large* as possible (possibly infinitely so) turning the capacity dial all the way to the right, and optimize its corresponding cost

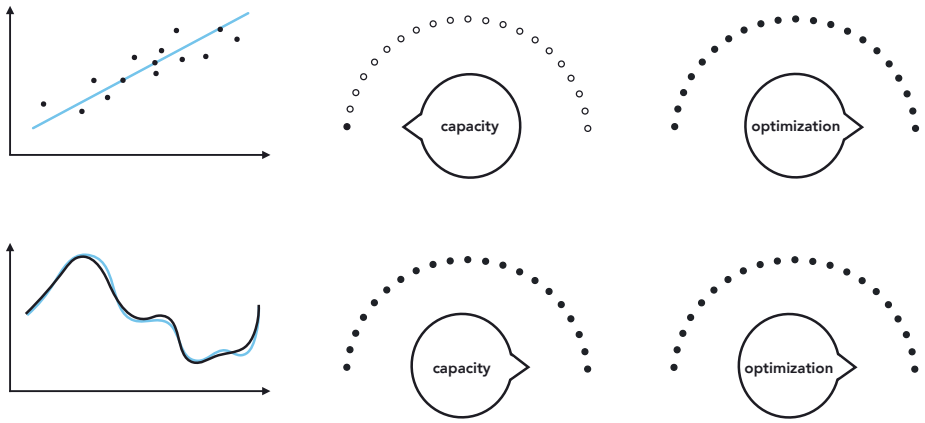

Figure 11.13 Model capacity and optimization precision visualized as two dials. When the capacity dial is set to the left we have a low-capacity linear model, when set to the right we admit maximum (perhaps infinite) capacity into the model. The optimization dial set to the left denotes the initial point of optimization, and all the way to the right denotes the final step of successful optimization. (top row) With linear regression (as we saw in previous chapters) we set the capacity dial all the way to the *left* and the optimization dial all the way to the *right* in order to find the best possible set of parameters for a low-capacity linear model (drawn in blue) that fits the given regression data. (bottom row) With universal approximation of a continuous function (drawn in black) we set both dials to the *right*, admitting infinite capacity into the model and tuning its parameters by optimizing to completion. See text for further discussion.

function as *well* as possible, turning the optimization dial all the way to the right as well.

In contrast, with the sort of linear learning we have looked at in previous chapters (as depicted in the top row of the Figure 11.13) we set our capacity dial all the way to the left (employing a linear model) but still set our optimization dial all the way to the right. By optimizing to completion we determine the proper bias and slope(s) of our linear model when performing, e.g., linear regression, as depicted in the figure.

We now examine a number of simple examples of universal approximation using various *near-perfect* regression and two-class classification datasets, where we set both the capacity and optimization dials far to the right. Here near-perfect means a very finely sampled, large dataset (as opposed to a perfect, infinitely large one). The case where a dataset is truly infinitely large ($P = \infty$) would, in theory, require infinite computing power to minimize a corresponding cost function.

Example 11.4 Universal approximation of near-perfect regression data

In Figure 11.14 we illustrate universal approximation of a near-perfect regres-

sion dataset consisting of $P = 10,000$ evenly sampled points from an underlying sinusoidal function defined over the unit interval. In the left, middle, and right columns we show the result of fitting an increasing number of polynomial, neural network, and tree units, respectively, to this data. As we increase the number of units in each case (from top to bottom) the capacity of each corresponding model increases, allowing for a better universal approximation.

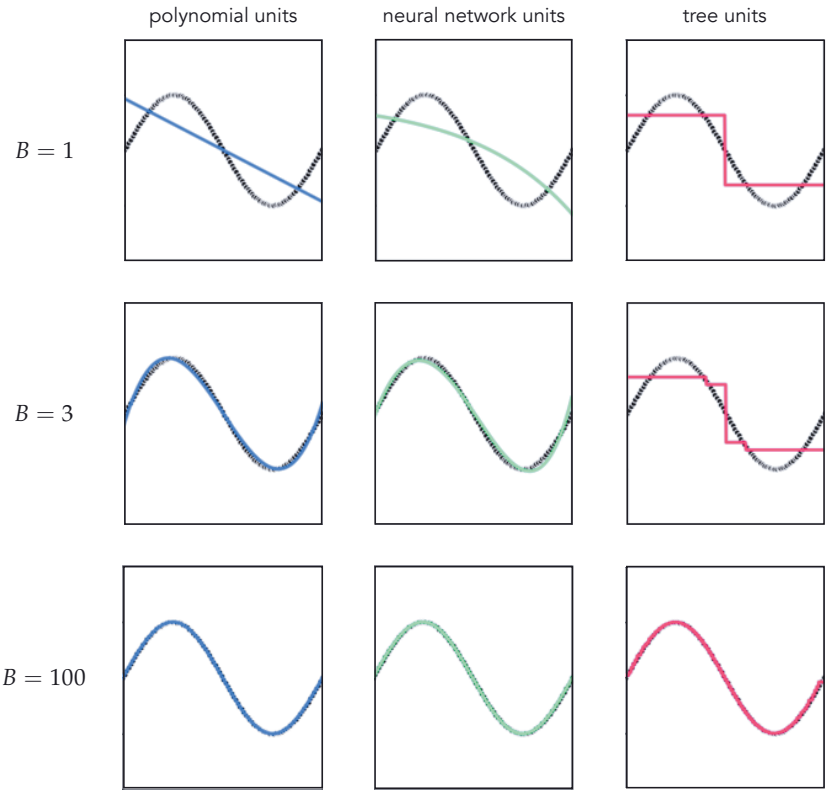

Figure 11.14 Figure associated with Example 11.4. Models built from polynomial (left column), neural network (middle column), and stump units (right column) fit to a near-perfect regression dataset. In visualizing the tree-based models in the right column we have connected each discrete step via a vertical line for visualization purposes only. As more and more units are added to the models each is able to fit the dataset with increasing precision. See text for further details.

Note here that it takes far fewer units of both the polynomial and neural network approximators to represent the data well as compared to the discrete stump units. This is because members of the former more closely resemble the smooth sinusoidal function that generated the data in the first place. This sort of phenomenon is true in general: while any type of universal approximator can be used to approximate a perfect (or near-perfect) dataset as closely as

desired, some universal approximators require fewer units to do so than the others depending on the shape of the underlying function that generated the dataset.

Example 11.5 Universal approximation of near-perfect classification data
In the top row of Figure 11.15 we show four instances of near-perfect two-class classification data from the perceptron perspective (i.e., from the top) each consisting of $P = 10,000$ points. In each instance those points colored red have label value +1, and those colored blue have label value −1. Plotted in the second row of this figure are the corresponding datasets shown from the regression perspective (i.e., from the side).

Each of these near-perfect datasets can be approximated effectively using any of the three catalogs of universal approximators discussed in Section 11.2.3, provided that the capacity of each model is increased sufficiently and that the corresponding parameters are tuned properly. In the third and fourth rows of the figure we show the resulting fit from employing $B = 30$ polynomial approximators using a Least Squares and Softmax cost, respectively.

11.3 Universal Approximation of Real Data

In the previous section we saw how a nonlinear model built from units of a single universal approximator can be made to tightly approximate any *perfect dataset* if we increase its capacity sufficiently and tune the model's parameters properly by minimizing an appropriate cost function. In this section we will investigate how universal approximation carries over to the case of *real data*, i.e., data that is finite in size and potentially noisy. We will then learn about a new measurement tool, called *validation error*, that will allow us to effectively employ universal approximators with real data.

11.3.1 Prototypical examples

Here we explore the use of universal approximators in representing real data using two simple examples: a regression and two-class classification dataset. The problems we encounter with these two simple examples mirror those we face in general when employing universal approximator based models with real data, regardless of problem type.

Example 11.6 Universal approximation of real regression data
In this example we illustrate the use of universal approximators on a real regression dataset that is based on the near-perfect sinusoidal data presented in Example 11.4. To simulate a real version of this dataset we randomly selected

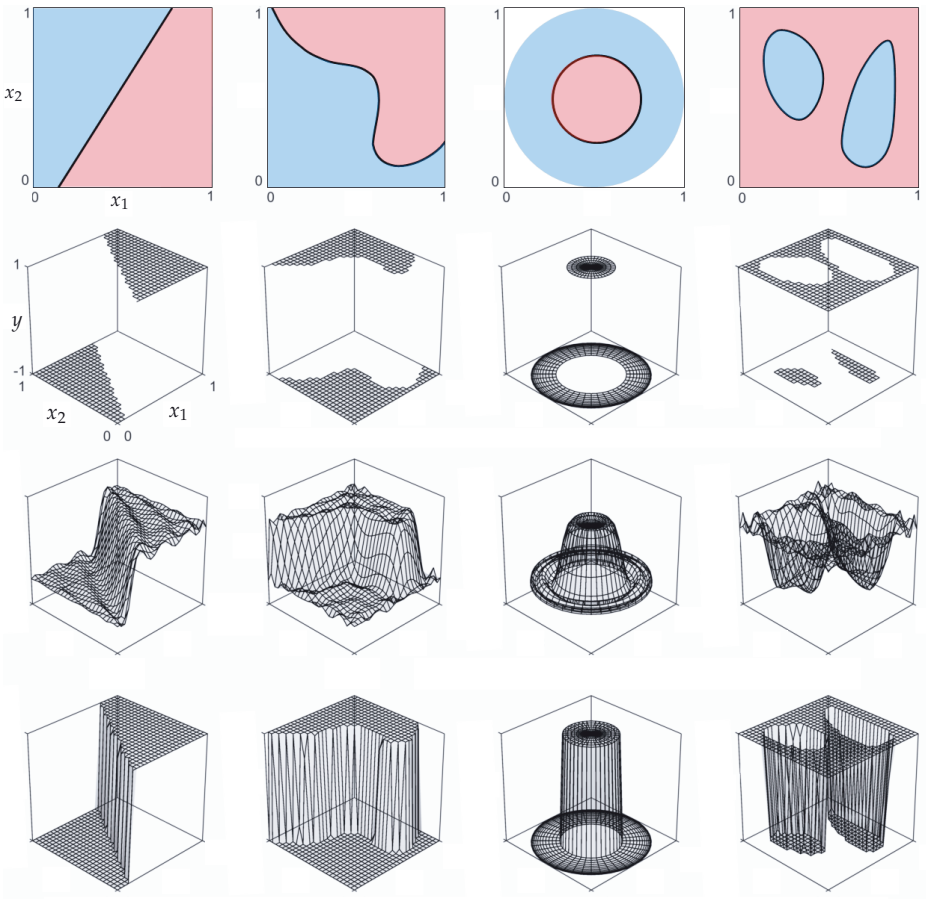

Figure 11.15 Figure associated with Example 11.5. (top row) Four instances of near-perfect two-class classification data. (second row) The corresponding data shown from a different (i.e., regression) perspective. Respective polynomial approximations to each dataset with $B = 30$ units in each instance using a Least Squares cost (third row) and a Softmax cost (fourth row). The approximations shown in the final row are passed through the tanh function before visualization. See text for further details.

$P = 21$ of its points and added a small amount of random noise to the output (i.e., y component) of each point, as illustrated in Figure 11.16.

In Figure 11.17 we illustrate the fully tuned nonlinear fit of a model employing polynomial (top row), neural network (middle row), and tree units (bottom row) to this data. Notice how, with each of the universal approximators, all three models *underfit* the data when using only $B = 1$ unit in each case (leftmost column). This underfitting of the data is a direct consequence of using low-capacity models, which produce fits that are not *complex* enough for the underlying data they are aiming to approximate. Also notice how each model improves as we increase B, but only up to a certain point after which each *tuned* model becomes far

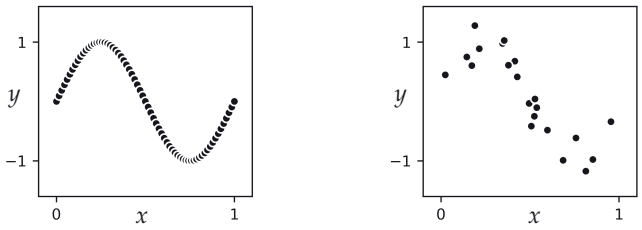

Figure 11.16 Figure associated with Example 11.6. (left panel) The original *near-perfect* sinusoidal dataset from Example 11.4. (right panel) A *real* regression dataset formed by adding random noise to the output of a small subset of the near-perfect dataset's points.

too complex and starts to look rather wild, and very much unlike the sinusoidal phenomenon that originally generated the data. This is especially visible in the polynomial and neural network cases, where by the time we reach $B = 20$ units (rightmost column) both models are extremely oscillatory and far too complex. Such *overfitting* models while representing the current data well, will clearly make for poor predictors of future data generated by the same process.

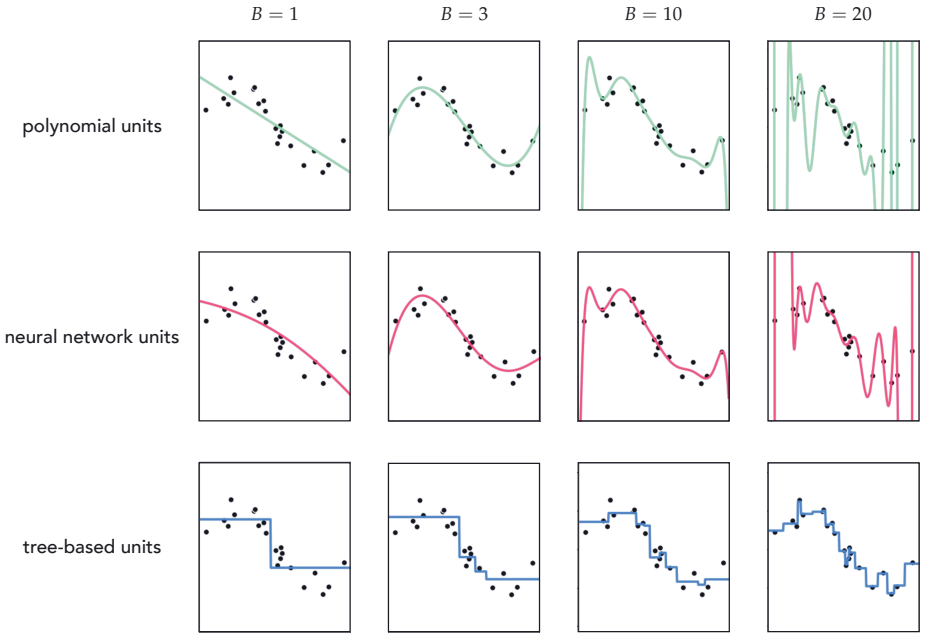

Figure 11.17 Figure associated with Example 11.6. See text for details.

In Figure 11.18 we plot several of the polynomial based models shown in Figure 11.17, along with the corresponding Least Squares cost value each attains. In adding more polynomial units we turn up the capacity of our model

and, optimizing each model to completion, the resulting tuned models achieve lower and lower cost value. However, the resulting fit provided by each fully tuned model (after a certain point) becomes far too complex and starts to get *worse* in terms of how it represents the general regression phenomenon. As a measurement tool the cost value only tells us how well a tuned model fits the *training data*, but fails to tell us when our tuned model becomes too complex.

Example 11.7 Universal approximation of real classification data

In this example we illustrate the application of universal approximator-based models on a real two-class classification dataset that is based on the near-perfect dataset presented in Example 11.5. Here we simulated a realistic version of this data by randomly selecting $P = 99$ of its points, and adding a small amount of classification noise by flipping the labels of five of those points, as shown in Figure 11.19.

In Figure 11.20 we show the nonlinear decision boundaries provided by fully tuned models employing polynomial (top row), neural network (middle row), and tree units (bottom row). In the beginning where $B = 2$ (leftmost column) all three tuned models are not complex enough and thus *underfit* the data, providing a classification that in all instances simply classifies the entire space as belonging to the blue class. After that and up to a certain point the decision boundary provided by each model improves as more units are added, with $B = 5$ polynomial units, $B = 3$ neural network units, and $B = 5$ tree units providing reasonable approximations to the desired circular decision boundary. However, soon after we reach these numbers of units each tuned model becomes too complex and *overfits* the training data, with the decision boundary of each drifting away from the true circular boundary centered at the origin. As with regression in Example 11.6, both underfitting and overfitting problems occur in the classification case as well, regardless of the sort of universal approximator used.

In Figure 11.21 we plot several of the neural network based models shown in the middle row of Figure 11.20, along with the corresponding two-class Softmax cost value each attains. As expected, increasing model capacity by adding more neural network units always (upon tuning the parameters of each model by complete minimization) *decreases* the cost function value (just as with perfect or near-perfect data). However, the resulting classification, after a certain point, actually gets *worse* in terms of how it (the learned decision boundary) represents the general classification phenomenon.

In summary, Examples 11.6 and 11.7 show that, unlike the case with perfect data, when employing universal approximator based models with real data we must be careful with how we set the capacity of our model, as well as how well we tune its parameters via optimization of an associated cost. These two

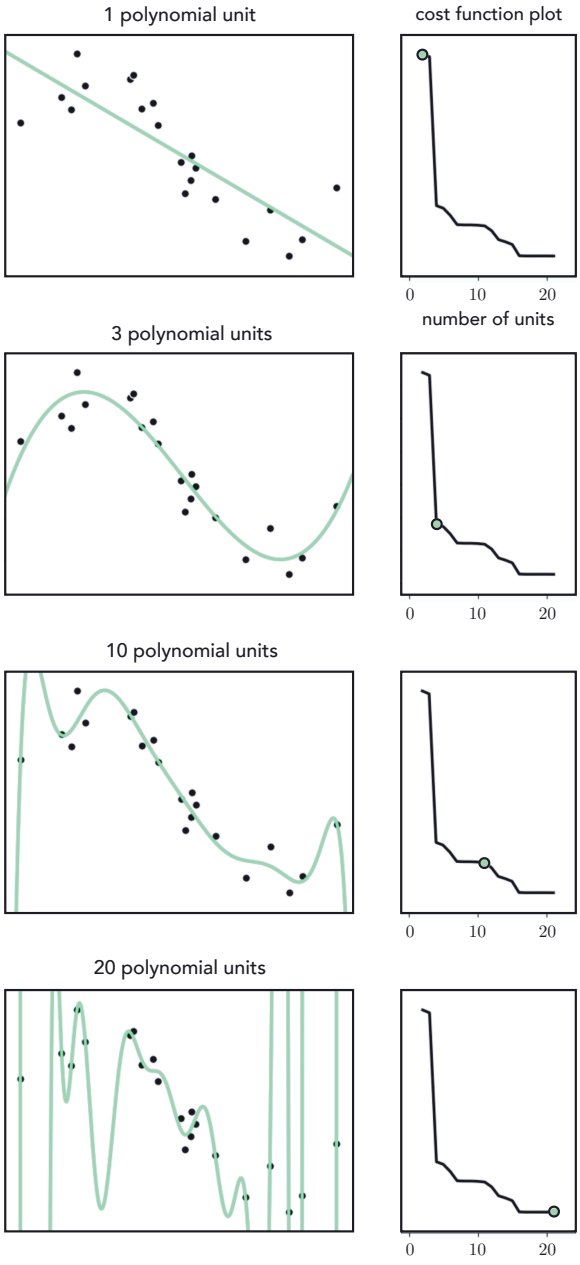

Figure 11.18 Figure associated with Example 11.6. See text for details.

simple examples also show how the cost value associated with training data (also called *training error*) fails as a reliable tool to measure how well a tuned

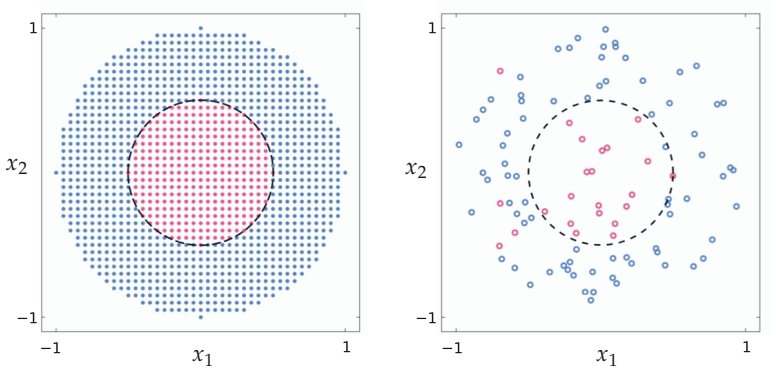

Figure 11.19 Figure associated with Example 11.7. (left panel) The original near-perfect classification dataset from Example 11.5, with the true circular boundary used to generate the data shown in dashed black (this is the boundary we hope to recover using classification). (right panel) A real dataset formed from a noisy subset of these points. See text for further details.

model represents the phenomenon underlying a real dataset. Both of these issues arise in general, and are discussed in greater detail next.

11.3.2 The capacity and optimization dials, revisited

The prototypical examples described in Section 11.3.1 illustrate how with real data we cannot (as we can in the case of perfect data) simply set our capacity and optimization dials (introduced in Section 11.2.4) all the way to the right, as this leads to overly complex models that fail to represent the underlying data-generating phenomenon well. Notice, we only control the *complexity* of a tuned model (or, roughly speaking, how "wiggly" a tuned model fit is) *indirectly* by how we set both our capacity and optimization dials, and it is not obvious *a priori* how we should set them simultaneously in order to achieve the right amount of model complexity for a given dataset. However, we can make this dial-tuning problem somewhat easier by fixing one of the two dials and adjusting only the other. Setting one dial all the way to the right imbues the other dial with the sole control over the complexity of a tuned model (and turns it into – roughly speaking – the *complexity dial* described in Section 11.1.3). That is, fixing one of the two dials all the way to the right, as we turn the unfixed dial from left to right we increase the complexity of our final tuned model. This is a general principle when applying universal approximator based models to real data that does not present itself in the case of perfect data.

To gain a stronger intuition for this principle, suppose first that we set our optimization dial all the way to the right (meaning that regardless of the dataset and model we use, we always tune its parameters by minimizing the corresponding cost function to completion). Then with perfect data, as illustrated in

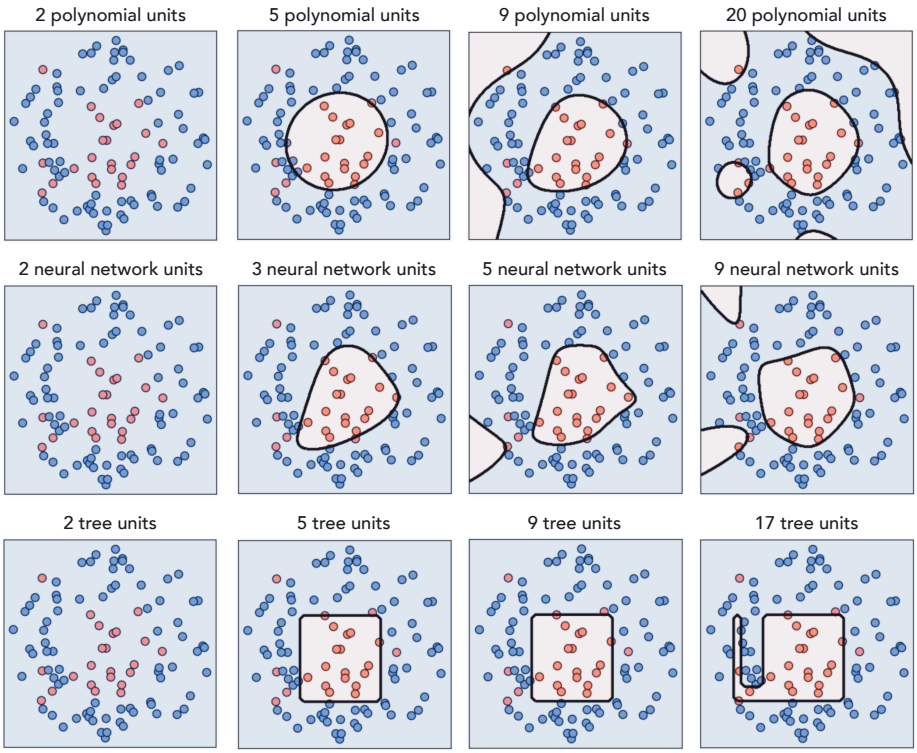

Figure 11.20 Figure associated with Example 11.7. See text for details.

the top row of Figure 11.22, as we turn our capacity dial from left to right (e.g., by adding more units) the resulting tuned model provides a better and better representation of the data.

However, with real data, as illustrated in the bottom row of Figure 11.22, starting with our capacity dial all the way to the left, the resulting tuned model is not complex enough for the phenomenon underlying our data. We say that such a tuned model *underfits*, as it does not fit the given data well.[7] Turning the capacity dial from left to right *increases* the complexity of each tuned model, providing a better and better representation of the data and the phenomenon underlying it. However, there comes a point, as we continue turning the dial from left to right, where the corresponding tuned model becomes *too complex*. Indeed past this point, where the complexity of each tuned model is wildly inappropriate for the phenomenon at play, we say that *overfitting* begins. This language is used because while such highly complex models fit the given data extremely well, they do so at the cost of not representing the underlying phenomenon well. As

[7] Notice that while the simple visual depiction here illustrates an underfitting model as a linear ("unwiggly") function – which is quite common in practice – it is possible for an underfitting model to be quite "wiggly." Regardless of the shape a tuned model takes, we say that it underfits if it poorly represents the training data, i.e., if it has high training error.

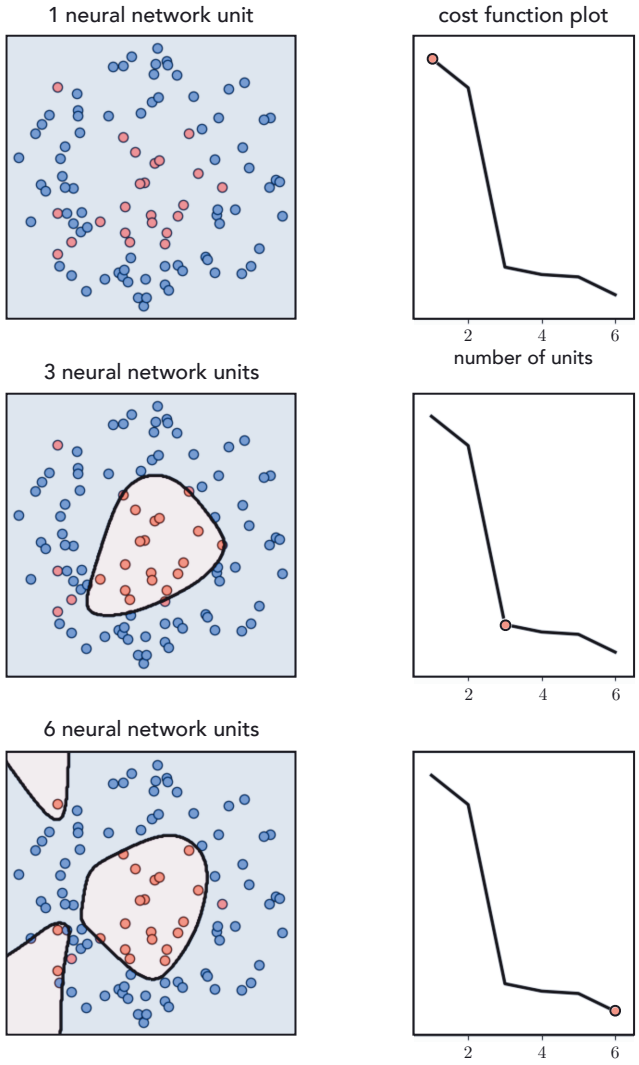

Figure 11.21 Figure associated with Example 11.7. See text for details.

we continue to turn our capacity dial to the right the resulting tuned models will become increasingly complex and increasingly less representative of the true underlying phenomenon.

Now suppose instead that we turn our capacity dial all the way to the right, using a very high-capacity model, and set its parameters by turning our optimization dial ever so slowly from left to right. In the case of perfect data, as illustrated in the top row of Figure 11.23, this approach produces tuned models that increasingly represent the data well. With real data on the other hand, as illustrated in the bottom row of Figure 11.23, starting with our optimization dial

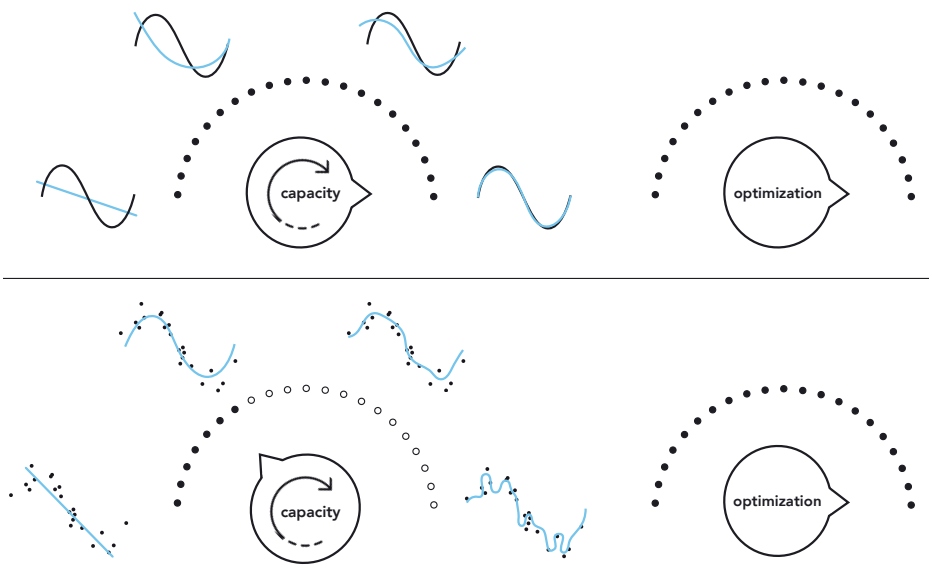

Figure 11.22 (top row) With *perfect data* if we set our optimization dial to all the way to the right, as we increase the capacity of our model by turning the capacity dial from left to right the corresponding representation gets better and better. (bottom row) With *real data* a similar effect occurs; however, here as we turn the capacity further to the right each tuned model will tend to become more and more complex, eventually overfitting the given data. See text for further details.

set all the way to the left will tend to produce low-complexity *underfitting* tuned models. As we turn our optimization dial from left to right, taking steps of a particular local optimization scheme, our corresponding model will tend to increase in complexity, improving its representation of the given data. This improvement continues only up to a point where our corresponding tuned model becomes too complex for the phenomenon underlying the data, and hence *overfitting* begins. After this point the tuned models arising from turning the optimization dial further to the right are far too complex to adequately represent the phenomenon underlying the data.

11.3.3 Motivating a new measurement tool

How we set our capacity and optimization dials in order to achieve a final tuned model that has *just the right amount of complexity* for a given dataset is the main challenge we face when employing universal approximator based models with real data. In Examples 11.6 and 11.7 we saw how training error fails to indicate when a tuned model has sufficient *complexity* for the tasks of regression and two-class classification, respectively – a fact more generally true about all nonlinear machine learning problems as well. If we cannot rely on training error to help decide on the proper amount of complexity required to address real nonlinear

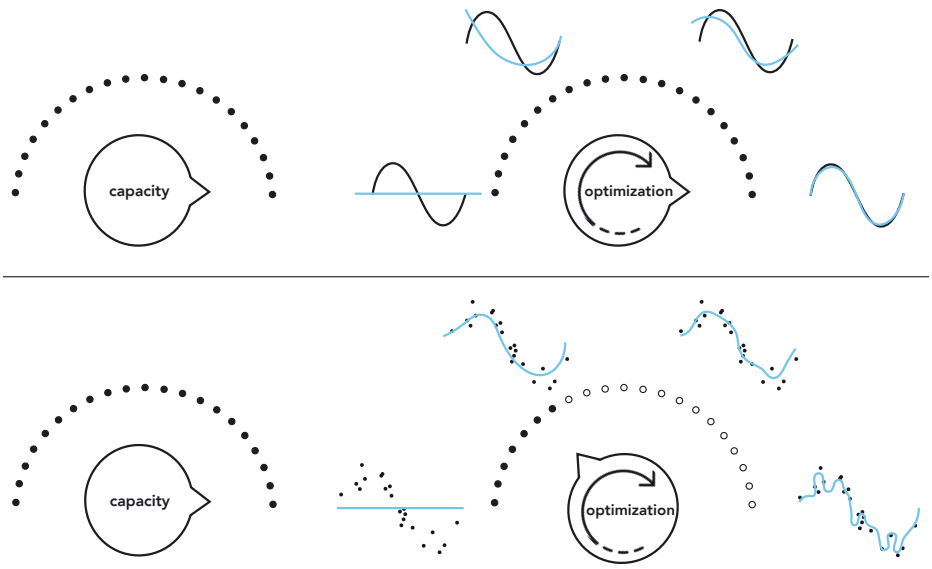

Figure 11.23 (top row) With *perfect data* if we set our capacity dial all the way to the right, as we increase the amount we optimize our model by turning the optimization dial from left to right (starting all the way on the left where for simplicity we assume all model parameters are initialized at zero) the corresponding representation gets better and better. (bottom row) With *real data* a similar effect occurs, but only up to a certain point where overfitting begins. See text for further details.

machine learning tasks, what sort of measurement tool should we use instead? Closely examining Figure 11.24 reveals the answer!

In the top row of this figure we show three instances of models presented for the toy regression dataset in Example 11.6: a fully tuned low-complexity (and underfitting) linear model in the left panel, a high-complexity (and overfitting) degree-20 polynomial model in the middle panel, and a degree-three polynomial model in the right panel that fits the data and the underlying phenomenon generating it "just right." What do both the underfitting and overfitting patterns have in common, that the "just right" model does not?

Scanning the left two panels of the figure we can see that a common problem with both the underfitting and overfitting models is that, while they differ in how well they represent data *we already have*, they will both fail to adequately represent *new data* generated via the same process by which the current data was made. In other words, we would not trust either model to predict the output of a newly arrived input point. The "just right" fully tuned model does not suffer from the same problem as it closely approximates the sort of wavy sinusoidal pattern underlying the data, and as a result would work well as a predictor for future data points.

The same story tells itself in the bottom row of Figure 11.24 with our two-

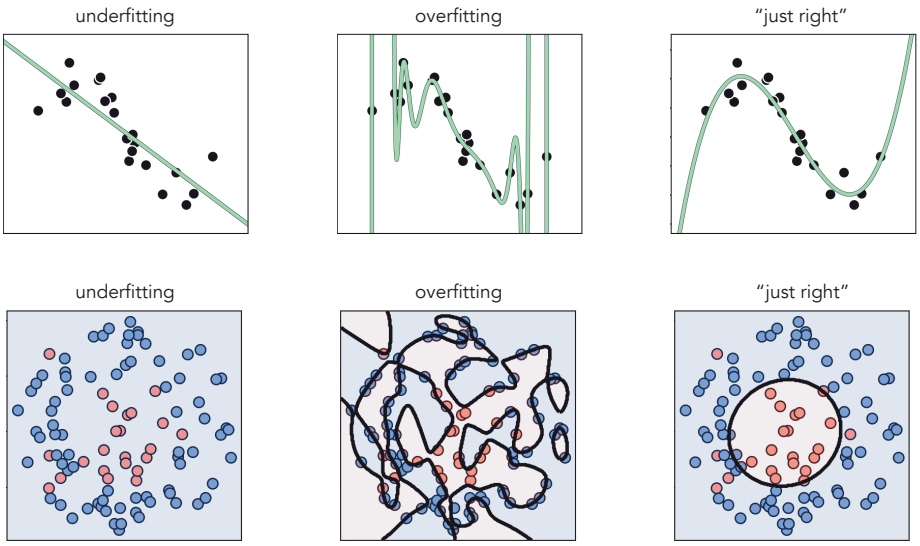

Figure 11.24 (top row) Three models for the regression dataset from Example 11.6: an underfitting model (top-left panel), an overfitting model (top-middle panel), and a "just right" one (top-right panel). (bottom row) Three models for the two-class classification dataset from Example 11.7: an underfitting model that simply classifies everything as part of the blue class (bottom-left panel), an overfitting model (bottom-middle panel), and a "just right" fit (bottom-right panel). See text for further details.

class classification dataset used previously in Example 11.7. Here we show a fully tuned low-complexity (and underfitting) linear model in the left panel, a high-complexity (and overfitting) degree-20 polynomial model in the middle panel, and a "just right" degree-two polynomial model in the right panel. As with the regression case, the underfitting and overfitting models both fail to adequately represent the underlying phenomenon generating our current data and, as a result, will fail to adequately predict the label values of *new data* generated via the same process by which the current data was made.

In summary, with both the simple regression and classification examples discussed here we can roughly qualify poorly-performing models as those that will not allow us to make accurate predictions of data we will receive in the future. But how do we quantify something we will receive in the future? We address this next.

11.3.4 The validation error

We now have an informal diagnosis for the problematic performance of underfitting/overfitting models: such models do not accurately represent new data we might receive in the future. But how can we make use of this diagnosis? We of course do not have access to any new data we will receive in the future. To

make this notion useful we need to translate it into a quantity we can always measure, regardless of the dataset/problem we are tackling or the kind of model we employ.

The universal way to do this is, in short, to *fake it*: we simply remove a random portion of our data and treat it as "new data we might receive in the future," as illustrated abstractly in Figure 11.25. In other words, we cut out a random chunk of the dataset we have, train our selection of models on only the portion of data that remains, and *validate* the performance of each model on this randomly removed chunk of "new" data. The random portion of the data we remove to validate our model(s) is commonly called the *validation data* (or validation set), and the remaining portion we use to train models is likewise referred to as the *training data* (or training set). The model providing the lowest error on the validation data, i.e., the lowest validation error, is then deemed the best choice from a selection of trained models. As we will see, validation error (unlike training error) is in fact a proper measurement tool for determining the quality of a model against the underlying data-generating phenomenon we want to capture.

Figure 11.25 Splitting the data into training and validation sets. The original data shown in the left panel as the entire round mass is split randomly in the right panel into two nonoverlapping sets. The smaller piece, typically $\frac{1}{10}$ to $\frac{1}{3}$ of the original data, is then taken as the validation set with the remaining taken as the training set.

There is no precise rule for what portion of a given dataset should be saved for validation. In practice, typically between $\frac{1}{10}$ to $\frac{1}{3}$ of the data is assigned to the validation set. Generally speaking, the larger and/or more representative (of the true phenomenon from which the data is sampled) a dataset is, the larger the portion of the original data may be assigned to the validation set (e.g., $\frac{1}{3}$). The intuition for doing this is that if the data is plentiful/representative enough, the training set still accurately represents the underlying phenomenon even after removal of a relatively large set of validation data. Conversely, in general with smaller or less representative (i.e., more noisy or poorly-distributed) datasets we usually take a smaller portion for validation (e.g., $\frac{1}{10}$) since the relatively larger training set needs to retain what little information of the underlying phenomenon was captured by the original data, and little data can be spared for validation.

11.4 Naive Cross-Validation

Validation error provides us with a concrete way of not only measuring the performance of a single tuned model, but more importantly it allows us to compare the efficacy of multiple tuned models of various levels of complexity. By carefully searching through a set of models ranging in complexity we can then easily identify the best of the bunch, the one that provides minimal error on the validation set. This comparison of models, called *cross-validation* or sometimes *model search* or *selection*, is the basis of feature learning as it provides a systematic way to *learn* (as opposed to *engineer*, as detailed in Chapter 10) the proper form a nonlinear model should take for a given dataset.

In this section we introduce what we refer to as *naive* cross-validation. This consists of a search over a set of models of varying capacity, with each model fully optimized over the training set, in search of a validation-error-minimizing choice. While it is simple in principle and in implementation, naive cross-validation is in general very expensive (computationally speaking) and often results in a rather coarse model search that can miss (or "skip over") the ideal amount of complexity desired for a given dataset.

11.4.1 The big picture

The first *organized* approach one might take to determining an ideal amount of complexity for a given dataset is to first choose a single universal approximator (e.g., one of those simple exemplars outlined in Section 11.2.3) and construct a set of M models of the general form given in Equation (11.15) by ranging the value of B from 1 to M sequentially as

$$\begin{aligned}
\text{model}_1\left(\mathbf{x}, \Theta_1\right) &= w_0 + f_1\left(\mathbf{x}\right) w_1 \\
\text{model}_2\left(\mathbf{x}, \Theta_2\right) &= w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 \\
&\vdots \\
\text{model}_M\left(\mathbf{x}, \Theta_M\right) &= w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 + \cdots + f_M\left(\mathbf{x}\right) w_M.
\end{aligned} \tag{11.16}$$

This set of models – which we can denote compactly as $\{\text{model}_m\left(\mathbf{x}, \Theta_m\right)\}_{m=1}^{M}$ (or even more compactly as just $\{\text{model}_m\}_{m=1}^{M}$) where the set Θ_m consists of all those parameters of the mth model – naturally increases in *capacity* from $m = 1$ to $m = M$ (as first described in Section 11.2.2). If we *optimize* every one of these models to completion they will also roughly speaking – as discussed in Section 11.3.2 – increase in terms of their *complexity* with respect to training data as well. Thus, if we first split our original data randomly into training and validation portions as detailed in Section 11.3.4, and measure the error of all M fully trained models on each portion of the data, we can very easily determine which of the M models provides the ideal amount of complexity for the dataset overall by finding the one that achieves minimum validation error.

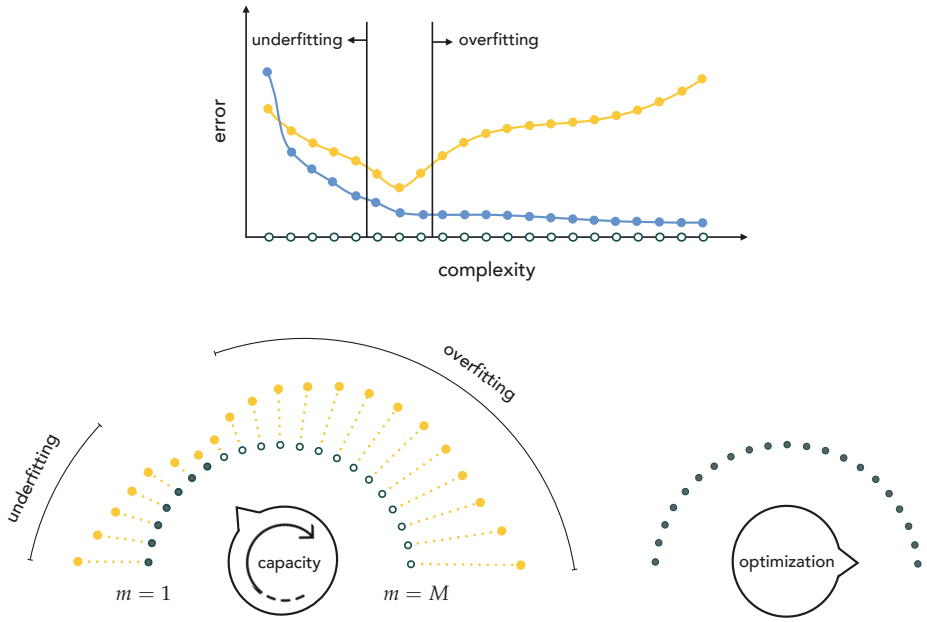

Figure 11.26 (top panel) Prototypical training (in blue) and validation (in yellow) error plots resulting from a run of naive cross-validation. Here the set of models – which generally increase in *complexity* with respect to the training set – are formed by fully optimizing a set of models of increasing *capacity*. Low-complexity models underfit the data, typically producing large training and validation errors. While the training error will monotonically decrease as model complexity increases, validation error tends to decrease only up to the point where overfitting of the training data begins. (bottom panels) Naive cross-validation using our dial conceptualization, where we turn the *capacity* dial from left to right, searching over a range of models of increasing capacity in search of a validation-error-minimizing model, while keeping the *optimization* dial set all the way to the right (indicating that we optimize each model to completion). See text for further details.

In the top panel of Figure 11.26 we show the generic sort of training (in blue) and validation (in yellow) errors we find in practice as a result of following this naive cross-validation scheme. The horizontal axis of this plot shows (roughly speaking) the complexity of each of our M fully optimized models, with the output on the vertical axis denoting error level. As can be seen in the figure, our low-complexity models *underfit* the data as reflected in their high training and validation errors. As the model complexity increases further, fully optimized models achieve lower training error since increasing model complexity allows us to constantly improve how well we can represent training data. This fact is reflected in the monotonically decreasing nature of the (blue) training error curve. On the other hand, while the validation error of our models will tend to decrease at first as we increase complexity, this trend continues only up to a point where *overfitting* of the training data begins. Once we reach a model

complexity that overfits the training data our validation error starts to increase again, as our model becomes less and less a fair representation of "data we might receive in the future" generated by the same phenomenon.

Note in practice that while training error typically follows the monotonically decreasing trend shown in the top panel of Figure 11.26, validation error can oscillate up and down more than once depending on the models tested. In any event, we determine the best fully optimized model from the set by choosing the one that *minimizes* validation error. This is often referred to as solving the *bias-variance trade-off*, as it involves determining a model that (ideally) neither underfits (or has high bias) nor overfits (or has high variance).

In the bottom row of Figure 11.26 we summarize this naive approach to cross-validation using the capacity/optimization dial conceptualization first introduced in Section 11.2.2. Here we set our *optimization* dial all the way to the right – indicating that we optimize every model to completion – and in ranging over our set of M models we turn the *capacity* dial from left to right starting with $m = 1$ (on the left) and ending with $m = M$ (all the way to the right), with the value of m increasing by 1 at each notch of the dial. Since in this case the *capacity* dial roughly governs model complexity – as summarized visually in the bottom row of Figure 11.22 – our model search reduces to setting this dial correctly to the minimum validation error setting. To visually denote how this is done we wrap the prototypical validation error curve shown in the top panel of Figure 11.26 clockwise around the capacity dial. We can then imagine setting this dial correctly (and automatically) to the value of m providing minimum validation error.

Example 11.8 Naive cross-validation and regression

In this example we illustrate the use of a naive cross-validation procedure on the sinusoidal regression dataset first introduced in Example 11.6. Here we use $\frac{2}{3}$ of the original set of 21 data points for training, and the remaining $\frac{1}{3}$ for validation. The set of models we compare here are polynomials of degree $1 \leq m \leq 8$. In other words, the mth model from our set $\{\text{model}_m\}_{m=1}^{8}$ is a single-input degree-m polynomial of the form

$$\text{model}_m\left(\mathbf{x}, \Theta_m\right) = w_0 + xw_1 + x^2w_2 + \cdots + x^m w_m. \tag{11.17}$$

Note how this small set of models is naturally ordered in terms of nonlinear capacity, with lower-degree models having smaller capacity and higher-degree models having larger capacity.

Figure 11.27 shows the fit of three polynomial models on the original dataset (first row), training data (second row), and validation data (third row). The errors on both the training (in blue) and validation (in yellow) data is shown in the bottom panel for all eight models. Notice, the validation error is at its lowest when the model is a degree-four polynomial. Of course as we use more poly-

nomial units, moving from left to right in the figure, the higher-degree models fit the training data better. However, as the training error continues to decrease, the corresponding validation error starts climbing rapidly as the corresponding models provide poorer and poorer representations of the validation data (by the time $m = 7$ the validation error becomes so large that we do not plot it in the same window so that the other error values can be distinguished properly).

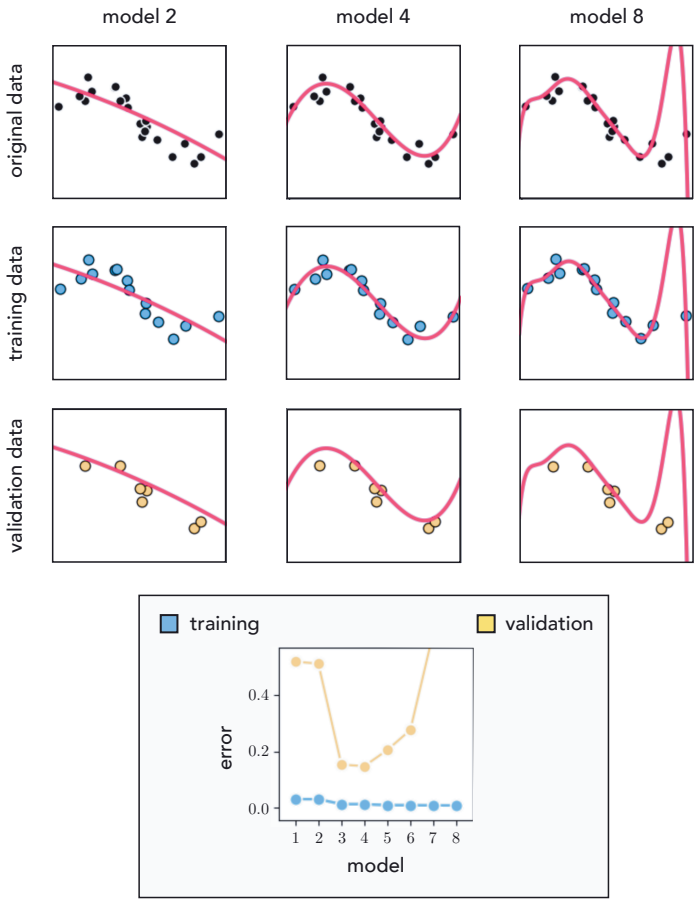

Figure 11.27 Figure associated with Example 11.8. See text for details.

Example 11.9 Naive cross-validation and classification

In this example we illustrate the use of a naive approach to cross-validation on the two-class classification dataset first shown in Example 11.7. Here we use (approximately) $\frac{4}{5}$ of the original set of 99 data points for training, and the other $\frac{1}{5}$ for validation. For the sake of simplicity we employ only a small set of poly-

nomial models having degrees $1 \leq m \leq 7$. In other words, the mth model from our set $\{\text{model}_m\}_{m=1}^{7}$ is a degree-m polynomial (with two-dimensional input) of the form

$$\text{model}_m\left(\mathbf{x}, \Theta_m\right) = w_0 + \sum_{0 < i+j \leq m} x_1^i x_2^j w_{i,j}. \tag{11.18}$$

These models are also naturally ordered from low to high capacity, as we increase the degree m of the polynomial.

Figure 11.28 shows the fit of three models from $\{\text{model}_m\}_{m=1}^{7}$ along with the original data (first row), the training data (second row), and the validation data (third row). The training and validation errors are likewise shown in the bottom panel for all seven models. With classification it makes more sense to use the number of misclassifications computed over the training/validation sets or some function of these misclassifications (e.g., accuracy) as our training/validation errors, as opposed to the raw evaluation of a classification cost.

In this case the degree-two polynomial model ($m = 2$) provides the smallest validation error, and hence the best nonlinear decision boundary for the entire dataset. This result does make intuitive sense as well, as we determined a circular boundary using a model of this form when engineering such features in Example 10.5 of Section 10.4.2. As the complexity goes up and training error continues to decrease, our models overfit the training data while at the same time providing a poor solution for the validation data.

11.4.2 Problems with naive cross-validation

Naive cross-validation works reasonably well for simple examples like those described above. However, since the process generally involves trying out a range of models where each model is optimized *completely* from scratch, naive cross-validation can be very expensive computationally speaking. Moreover, the *capacity* difference between even adjacent models (e.g., those consisting of m and $m + 1$ units) can be quite large, leading to huge jumps in the range of model complexities tried out on a dataset. In other words, controlling model *complexity* via adjustment of the *capacity* dial (with our *optimization* dial turned all the way to the right – as depicted in the bottom panels of Figure 11.26) often only allows for a coarse model search that can easily "skip over" an ideal amount of model complexity. As we will see in the next two sections, much more robust and fine-grained cross-validation schemes can be constructed by setting our *capacity* dial to the right and controlling model *complexity* by carefully setting our *optimization* dial.

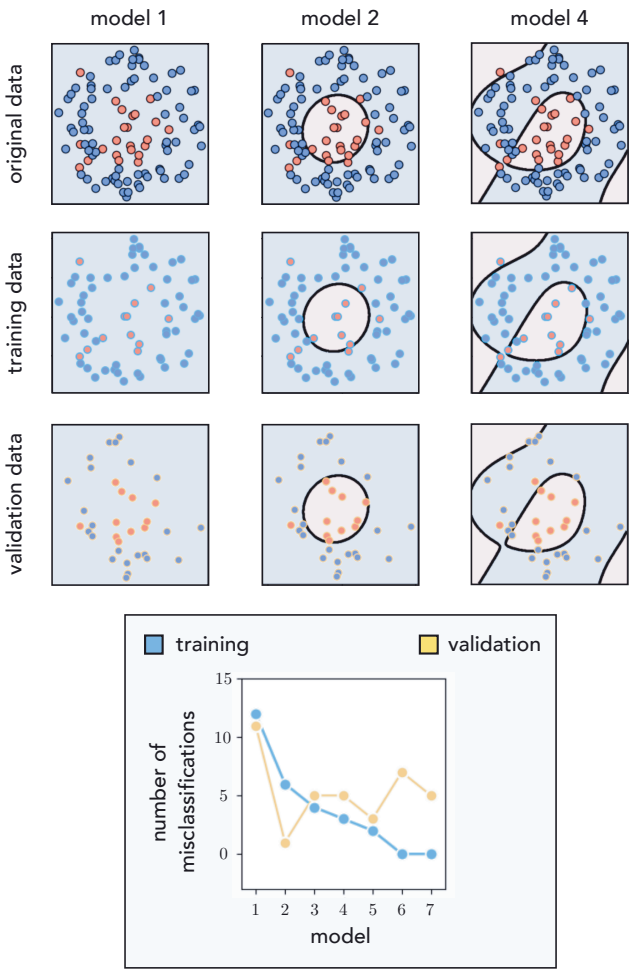

Figure 11.28 Figure associated with Example 11.9. See text for details.

11.5 Efficient Cross-Validation via Boosting

In this section we introduce *boosting*, the first of two fundamental paradigms for effective cross-validation described in this chapter. In contrast to the naive form of cross-validation described in the previous section, with boosting-based cross-validation we perform our model search by taking a single high-capacity model and optimize it *one unit at a time*, resulting in a much more efficient cross-validation procedure. While in principle any universal approximator can be used with boosting, this approach is often used as the cross-validation method of choice when employing tree-based universal approximators (as discussed further in Section 14.7).

11.5.1 The big picture

The basic principle behind boosting-based cross-validation is to progressively build a high-capacity model *one unit at a time*, using units from a single type of universal approximator (e.g., one of those simple exemplars outlined in Section 11.2.3), as

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 + \cdots + f_M\left(\mathbf{x}\right) w_M. \tag{11.19}$$

We do this sequentially in M *rounds*[8] where at each round we add one unit to the model, completely optimizing this unit's parameters alone along with its corresponding linear combination weight, and keep these parameters fixed at these optimally tuned values forever more. Alternatively, we can think of this procedure as beginning with a high-capacity model of the form in Equation (11.19) and – in M rounds – optimizing the parameters of each unit, one at a time.[9] In either case, performing boosting in this way produces a sequence of M tuned models that generally increase in *complexity* with respect to the training dataset, which we denote compactly as $[\text{model}_m]_{m=1}^{M}$ where the mth model consists of m tuned units. Since just one unit is optimized at a time, boosting tends to provide a computationally efficient fine-resolution form of model search (compared to naive cross-validation).

The general boosting procedure tends to produce training/validation error curves that generally look like those shown in the top panel of Figure 11.29. As with the naive approach detailed in the previous section, here too we tend to see training error decrease as m grows larger while validation error tends to start high where underfitting occurs, dip down to a minimum value (perhaps oscillating more than the one time illustrated here), and rise back up when overfitting begins.

Using the capacity/optimization dial conceptualization first introduced in Section 11.2.4, we can think about boosting as starting with our *capacity dial* set all the way to the *right* at some high value (e.g., some large value of M), and fidgeting with the *optimization dial* by turning it very slowly from left to right, as depicted in the bottom row of Figure 11.29. As discussed in Section 11.3.2 and summarized visually in the bottom row of Figure 11.23, with real data this general configuration allows our *optimization dial* to govern model complexity. In other words, with this configuration our optimization dial (roughly speaking) becomes the sort of fine-resolution *complexity dial* we aimed to construct at the outset of the chapter (see Section 11.1.3). With our optimization dial turned all the way to the left we begin our search with a low-complexity tuned model (called model_1) consisting of a single unit of a universal approximator having its parameters fully optimized. As we progress through rounds of boosting we turn the optimization dial gradually from left to right (here each notch on the

[8] $M + 1$ rounds if we include w_0.

[9] This is a form of coordinate-wise optimization. See, for example, Section 2.6.

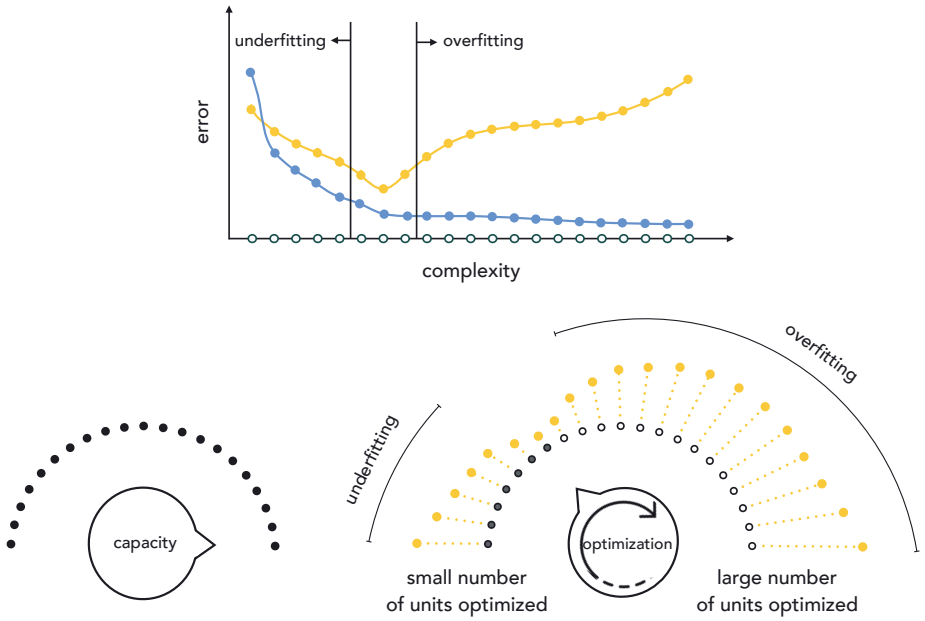

Figure 11.29 (top panel) Prototypical training and validation error curves associated with a completed run of boosting. (bottom panels) With boosting we fix our *capacity dial* all the way to the right, and begin with our *optimization dial* set all the way to the left. We then slowly turn our optimization dial from left to right, with each notch on the optimization dial denoting the complete optimization of one additional unit of the model, increasing the complexity of each subsequent model created with respect to the training set. See text for further details.

optimization dial denotes the complete optimization of one additional unit) optimizing (to completion) a single weighted unit of our original high-capacity model in Equation (11.19), so that at the mth round our tuned model (called model$_m$) consists of m individually but fully tuned units. Our ultimate aim in doing this is of course to determine a setting of the optimization (i.e., determine an appropriate number of tuned units) that minimizes validation error.

Whether we use fixed-shape, neural network, or tree-based units with boosting, we will naturally prefer units with *low capacity* so that the resolution of our model search is as fine-grained as possible. When we start adding units one at a time we turn our optimization dial clockwise from left to right. We want this dial turning to be done as smoothly as possible so that we can scan the validation error curve in a fine-grained fashion, in search of its minimum. This is depicted in the left panel of Figure 11.30. If we use *high-capacity* units at each round of boosting the resulting model search will be much coarser, as adding each additional unit results in aggressively turning the dial from left to right leaving large gaps in our model search, as depicted in the right panel of Figure 11.30. This kind of low-resolution search could easily result in us skipping over

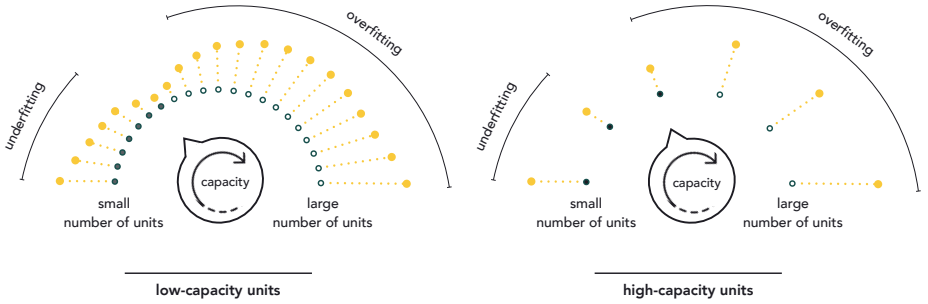

Figure 11.30 (left panel) Using low-capacity units makes the boosting procedure a high- (or fine-) resolution search for optimal model complexity. (right panel) Using high-capacity units makes boosting a low- (or coarse-) resolution search for optimal model complexity. See text for further details.

the complexity of an optimal model. The same can be said as to why we add only one unit at a time with boosting, tuning its parameters alone at each round. If we added more than one unit at a time, or if we retuned *every* parameter of *every* unit at each step of this process, not only would we have significantly more computation to perform at each step but the performance difference between subsequent models could be quite large and we might easily miss out on an ideal model.

11.5.2 Technical details

Formalizing our discussion of boosting above, we begin with a set of M nonlinear features or units from a single family of universal approximators

$$\mathcal{F} = \{f_1(\mathbf{x}),\, f_2(\mathbf{x}),\, \ldots,\, f_M(\mathbf{x})\}. \tag{11.20}$$

We add these units sequentially (or one at a time) building a sequence of M tuned models $[\text{model}_m]_{m=1}^{M}$ that increase in complexity with respect to the training data, from $m = 1$ to $m = M$, ending with a generic nonlinear model composed of M units. We will express this final boosting-made model slightly differently than in Equation (11.19), in particular reindexing the units it is built from as

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_{s_1}(\mathbf{x})\, w_1 + f_{s_2}(\mathbf{x})\, w_2 + \cdots + f_{s_M}(\mathbf{x})\, w_M. \tag{11.21}$$

Here we have reindexed the individual units to f_{s_m} to denote the unit from the entire collection in $\mathcal{F}$ added at the mth round of the boosting process. The linear combination weights w_0 through w_M along with any additional weights internal to $f_{s_1}, f_{s_2}, \ldots, f_{s_M}$ are represented collectively in the weight set Θ.

The process of boosting is performed in a total of M rounds, one for each of the units in Equation (11.21). At each round we determine which unit, when

added to the running model, best lowers its training error. We then measure the corresponding validation error provided by this update, and in the end after all rounds of boosting are complete, use the lowest validation error measurement found to decide which round provided the best overall model.

For the sake of simplicity in describing the formal details of boosting, we will center our discussion on a single problem: nonlinear regression on the training dataset $\left\{\left(\mathbf{x}_p,\, y_p\right)\right\}_{p=1}^{P}$ employing the Least Squares cost. However, the principles of boosting we will see remain *exactly* the same for other learning tasks (e.g., two-class and multi-class classification) and their associated costs.

Round 0 of boosting

We begin the boosting procedure by tuning

$$\text{model}_0\left(\mathbf{x}, \Theta_0\right) = w_0 \tag{11.22}$$

whose weight set $\Theta_0 = \{w_0\}$ contains a single bias weight, which we can easily tune by minimizing an appropriate cost over this variable alone. With this in mind, to find the optimal value for w_0 we minimize the Least Squares cost

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0\left(\mathbf{x}_p, \Theta_0\right) - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_0 - y_p\right)^2. \tag{11.23}$$

This gives the optimal value for w_0, which we denote as $w_0^\star$. We fix the bias weight at this value forever more throughout the process.

Round 1 of boosting

Having tuned the only parameter of model_0 we now *boost* its complexity by adding the weighted unit $f_{s_1}\left(\mathbf{x}\right) w_1$ to it, resulting in a modified running model which we call model_1

$$\text{model}_1\left(\mathbf{x}, \Theta_1\right) = \text{model}_0\left(\mathbf{x}, \Theta_0\right) + f_{s_1}\left(\mathbf{x}\right) w_1. \tag{11.24}$$

Note here the parameter set Θ_1 contains w_1 and any parameters internal to the unit f_{s_1}. To determine which unit in our set $\mathcal{F}$ best lowers the training error, we press model_1 against the data by minimizing

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0\left(\mathbf{x}_p, \Theta_0\right) + f_{s_1}\left(\mathbf{x}_p\right) w_1 - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_0^\star + f_{s_1}\left(\mathbf{x}_p\right) w_1 - y_p\right)^2 \tag{11.25}$$

for every unit $f_{s_1} \in \mathcal{F}$.

Note that since the bias weight has already been set optimally in the previous round we only need tune the weight w_1 as well as the parameters internal to the nonlinear unit f_{s_1}. Also note, in particular, that with neural networks all nonlinear units take precisely the same form, and therefore we need not solve M versions of the optimization problem in Equation (11.25), one for every unit in $\mathcal{F}$, as we would do when using fixed-shape or tree-based units. Regardless of the type of universal approximator employed, round 1 of boosting ends upon finding the optimal f_{s_1} and w_1, which we denote respectively as $f_{s_1}^\star$ and $w_1^\star$, and keep fixed moving forward.

Round $m > 1$ of boosting

In general, at the mth round of boosting we begin with model_{m-1} consisting of a bias term and $m-1$ units of the form

$$\text{model}_{m-1}\left(\mathbf{x}, \Theta_{m-1}\right) = w_0^\star + f_{s_1}^\star\left(\mathbf{x}\right) w_1^\star + f_{s_2}^\star\left(\mathbf{x}\right) w_2^\star + \cdots + f_{s_{m-1}}^\star\left(\mathbf{x}\right) w_{m-1}^\star. \quad (11.26)$$

Note that the parameters of this model have been tuned sequentially, starting with the bias $w_0^\star$ in round 0, $w_1^\star$ and any internal parameters of $f_{s_1}^\star$ in round 1, and so forth, up to $w_{m-1}^\star$ and any parameters internal to $f_{s_{m-1}}^\star$ in round $m-1$.

The mth round of boosting then follows the same pattern outlined in round 1, where we seek out the best weighted unit $f_{s_m}\left(\mathbf{x}\right) w_m$ to add to our running model to best lower its training error on the dataset. Specifically, our mth model takes the form

$$\text{model}_m\left(\mathbf{x}, \Theta_m\right) = \text{model}_{m-1}\left(\mathbf{x}, \Theta_{m-1}\right) + f_{s_m}\left(\mathbf{x}\right) w_m \quad (11.27)$$

and we determine the proper unit to add to this model by minimizing

$$\begin{aligned} &\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + f_{s_m}\left(\mathbf{x}_p\right) w_m - y_p\right)^2 \\ &= \frac{1}{P}\sum_{p=1}^{P}\left(w_0^\star + w_1^\star f_{s_1}^\star + \cdots + f_{s_{m-1}}^\star\left(\mathbf{x}_p\right) w_{m-1}^\star + f_{s_m}\left(\mathbf{x}_p\right) w_m - y_p\right)^2 \end{aligned} \quad (11.28)$$

over w_m and parameters internal to f_{s_m} (if they exist), which are contained in the parameter set Θ_m.

Once again with fixed-shape or tree-based approximators, this entails solving M (or $M - m + 1$, if we decide to check only those units not used in previous rounds) such optimization problems, and choosing the one with smallest training error. With neural networks, since each unit takes the same form, we need only solve one such optimization problem.

11.5.3 Early stopping

Once all rounds of boosting are complete note how we have generated a sequence of M tuned models[10] – denoted $\left[\text{model}_m\left(\mathbf{x}, \Theta_m\right)\right]_{m=1}^{M}$ – which gradually increase in nonlinear complexity from $m = 1$ to $m = M$, and thus gradually decrease in training error. This gives us fine-grained control in selecting an appropriate model, as the jump in performance in terms of both the training and validation errors between subsequent models in this sequence can be quite smooth, provided we use low-capacity units (as discussed in Section 11.5.1).

Once boosting is complete we select from our set of models the one that provides the lowest validation error. Alternatively, instead of running all rounds of boosting and deciding on an optimal model after the fact, we can attempt to *halt* the procedure when the validation error first starts to increase. This concept, referred to as *early stopping*, leads to a more computationally efficient implementation of boosting, but one needs to be careful in deciding when the validation error has really reached its minimum as it can oscillate up and down multiple times (as mentioned in Section 11.4), and need not take the simple generic form illustrated in the top panel of Figure 11.29. There is no ultimate solution to this issue – thus ad hoc solutions are typically used in practice when early stopping is employed.

11.5.4 An inexpensive but effective enhancement

A slight adjustment at each round of boosting, in the form of addition of an individual bias, can significantly improve the algorithm. Formally, at the mth round of boosting instead of forming model_m as shown in Equation (11.27), we add an additional bias weight $w_{0,m}$ as

$$\text{model}_m\left(\mathbf{x}, \Theta_m\right) = \text{model}_{m-1}\left(\mathbf{x}, \Theta_{m-1}\right) + w_{0,m} + f_{s_m}\left(\mathbf{x}\right) w_m. \tag{11.29}$$

This simple adjustment results in greater flexibility and generally better overall performance by allowing units to be adjusted "vertically" at each round (in the case of regression) at the minimal cost of adding a single variable to each optimization subproblem. Note that once tuning is done, the optimal bias weight $w_{0,m}^{\star}$ can be absorbed into the bias weights from previous rounds, creating a single bias weight $w_0^{\star} + w_{0,1}^{\star} + \cdots + w_{0,m}^{\star}$ for the entire model.

This enhancement is particularly useful when using fixed-shape or neural network units for boosting, as it is redundant when using tree-based approximators because they already have individual bias terms baked into them that always allow for this kind of vertical adjustment at each round of boosting.[11]

[10] We have excluded model_0 as it does not use any universal approximator units.

[11] In the jargon of machine learning boosting with tree-based learners is often referred to as *gradient boosting*. See Section 14.5 for further details.

To see this note that while Equation (11.14) shows the most common way of expressing a stump taking in one-dimensional input, repeated for convenience here, as

$$f(x) = \begin{cases} v_1 & x \leq s \\ v_2 & x > s \end{cases} \tag{11.30}$$

it is also possible to express $f(x)$ equivalently as

$$f(x) = w_0 + w_1 h(x) \tag{11.31}$$

where w_0 denotes an individual bias parameter for the stump and w_1 is an associated weight that scales $h(x)$, which is a simple step function with fixed levels and a split at $x = s$

$$h(x) = \begin{cases} 0 & x \leq s \\ 1 & x > s. \end{cases} \tag{11.32}$$

Expressing the stump in this equivalent manner allows us to see that every stump unit does indeed have its own individual bias parameter, making it redundant to add an individual bias at each round when boosting with stumps. The same concept holds for stumps taking in general N-dimensional input as well.

Example 11.10 Boosting regression using tree units

In this example we use the sinusoidal regression dataset first shown in Example 11.6 consisting of $P = 21$ data points, and construct a set of $B = 20$ tree (stump) units for this dataset (see Section 11.2.3). In Figure 11.31 we illustrate the result of $M = 50$ rounds of boosting (meaning many of the stumps are used multiple times). We split the dataset into $\frac{2}{3}$ training and $\frac{1}{3}$ validation, which are color-coded in light blue and yellow, respectively. Depicted in the figure are resulting regression fits and associated training/validation errors for several rounds of boosting. This example is discussed further in Section 14.5.

Example 11.11 Boosting classification using neural network units

In this example we illustrate the same kind of boosting as previously shown in Example 11.10, but now for two-class classification using a dataset of $P = 99$ data points that has a (roughly) circular decision boundary. This dataset was first used in Example 11.7. We split the data randomly into $\frac{2}{3}$ training and $\frac{1}{3}$ validation, and employ neural network units for boosting. In Figure 11.32 we illustrate the result of $M = 30$ rounds of boosting in terms of the nonlinear decision boundary and resulting classification, as well as training/validation errors.

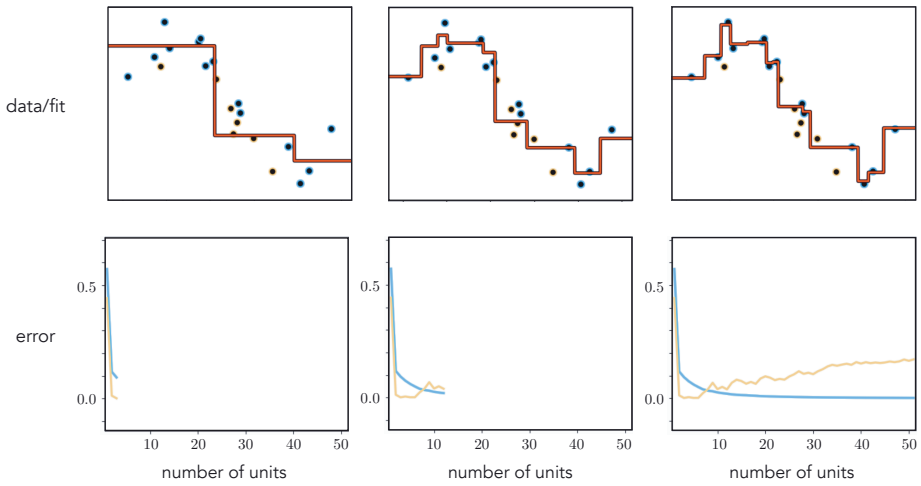

Figure 11.31 Figure associated with Example 11.10. See text for details.

11.5.5 Similarity to feature selection

The careful reader will notice how similar the boosting procedure is to the one introduced in Section 9.6 in the context of feature selection. Indeed principally the two approaches are entirely similar, except with boosting we do not select from a set of given input features but create them ourselves based on a chosen universal approximator family. Additionally, unlike feature selection where our main concern is *human interpretability*, we primarily use boosting as a tool for cross-validation. This means that unless we specifically prohibit it from occurring, we can indeed select the same feature multiple times in the boosting process as long as it contributes positively towards finding a model with minimal validation error.

These two use-cases for boosting, i.e., feature selection and cross-validation, can occur together, albeit typically in the context of linear modeling as detailed in Section 9.6. Often in such instances cross-validation is used with a linear model as a way of automatically selecting an appropriate number of features, with human interpretation of the resulting selected features still in mind. On the other hand, rarely is feature selection done when employing a nonlinear model based on features from a universal approximator due to the great difficulty in the human interpretability of nonlinear features. The rare exception to this rule is when using tree-based units which, due to their simple structure, can in particular instances be readily interpreted by humans.

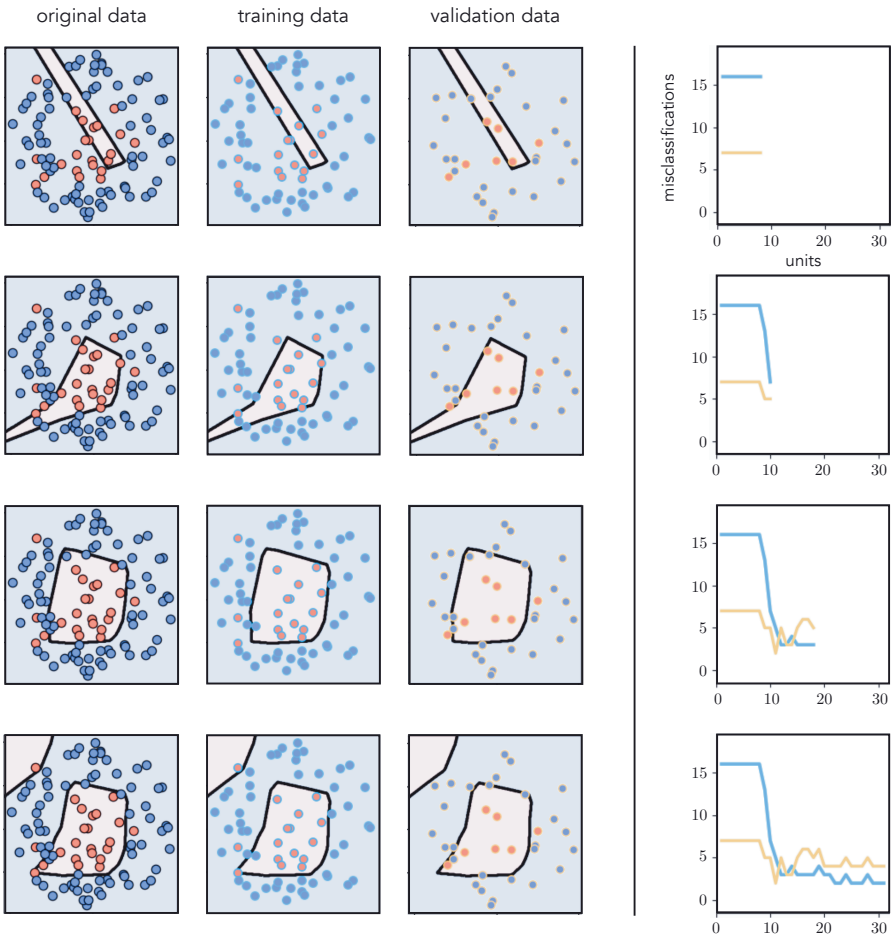

Figure 11.32 Figure associated with Example 11.11. See text for details.

11.5.6 The residual perspective with regression

Here we describe a common interpretation of boosting in the context of regression, that of sequentially fitting to the *residual* of a regression dataset. To see what this means, consider the following Least Squares cost function where we have inserted a boosted model at the mth round of its development

$$g\left(\Theta_m\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_m\left(\mathbf{x}_p, \Theta_m\right) - y_p\right)^2. \tag{11.33}$$

We can write our boosted model recursively as

$$\text{model}_m\left(\mathbf{x}_p, \Theta_m\right) = \text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + f_m\left(\mathbf{x}_p\right) w_m \tag{11.34}$$

where all of the parameters of the $(m-1)$th model (i.e., model_{m-1}) are already tuned. Combining Equations (11.33) and (11.34) we can rewrite the Least Squares cost as

$$g\left(\Theta_m\right) = \frac{1}{P}\sum_{p=1}^{P}\left(f_m\left(\mathbf{x}_p\right)w_m - \left(y_p - \text{model}_{m-1}\left(\mathbf{x}_p\right)\right)\right)^2. \tag{11.35}$$

By minimizing this cost we look to tune the parameters of a single additional unit so that

$$f_m\left(\mathbf{x}_p\right)w_m \approx y_p - \text{model}_{m-1}\left(\mathbf{x}_p\right) \tag{11.36}$$

for all p or, in other words, so that this fully tuned unit approximates our original output y_p minus the contribution of the previous model. This quantity, the difference between our original output and the contribution of the $(m-1)$th model, is often called the *residual*: it is what is left to represent after subtracting off what was learned by the $(m-1)$th model.

Example 11.12 Boosting from the perspective of fitting to the residual

In Figure 11.33 we illustrate the process of boosting $M = 20$ neural network units to a toy regression dataset. In the top panels we show the dataset along with the fit provided by model_m at the mth step of boosting for select values of m. In the corresponding bottom panels we plot the *residual* at the same step, as well as the fit provided by the corresponding mth unit f_m. As boosting progresses, the fit on the original data improves while (simultaneously) the residual shrinks.

11.6 Efficient Cross-Validation via Regularization

In the previous section we saw how with boosting based cross-validation we automatically learn the proper level of model complexity for a given dataset by optimizing a general high-capacity model one unit at a time. In this section we introduce what are collectively referred to as *regularization* techniques for efficient cross-validation. With this set of approaches we once again start with a single high-capacity model, and once again adjust its complexity with respect to a training dataset via careful optimization. However, with regularization we tune all of the units *simultaneously*, controlling how well we *optimize* its associated cost so that a minimum validation instance of the model is achieved.

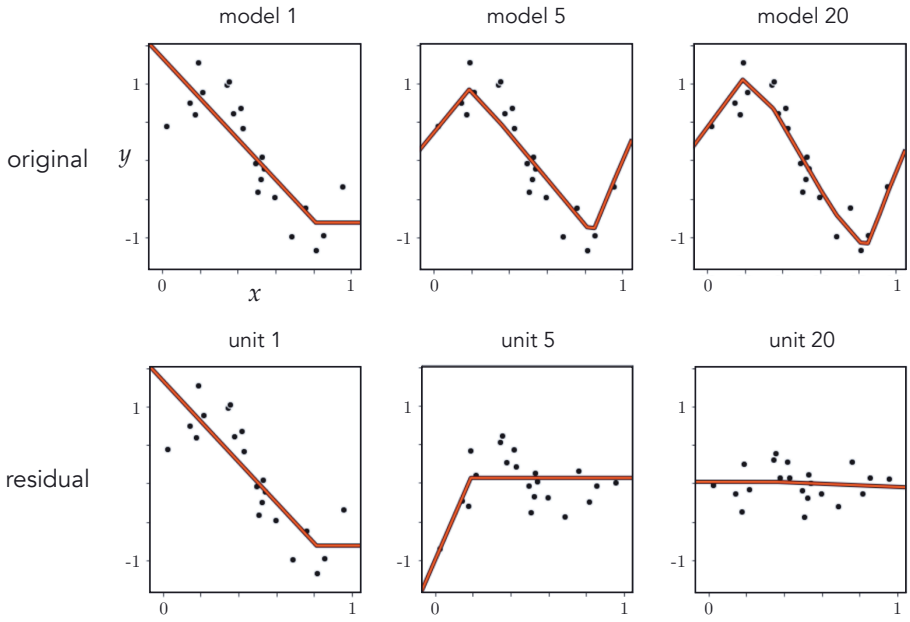

Figure 11.33 Figure associated with Example 11.12. See text for details.

11.6.1 The big picture

Imagine for a moment that we have a simple nonlinear regression dataset, like the one shown in the top-left panel of the Figure 11.34, and we use a high-capacity model (relative to the nature of the data) made up of a sum of *universal approximators* of a single kind to fit it as

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 + \cdots + f_M\left(\mathbf{x}\right) w_M. \tag{11.37}$$

Suppose that we partition this data into training and validation portions, and then train our high-capacity model by *completely* optimizing the Least Squares cost over the training portion of the data. In other words, we determine a set of parameters for our high-capacity model that lie very close to a global minimum of its associated cost function. In the top-right panel of the figure we draw a hypothetical two-dimensional illustration of the cost function associated with our high-capacity model over the training data, denoting the global minimum by a blue dot and its evaluation on the function by a blue $\mathbf{x}$.

Since our model has high capacity, the resulting fit provided by the parameters lying at the global minimum of our cost will produce a tuned model that is overly complex and *severely* overfits the training portion of our dataset. In the bottom-left panel of the Figure 11.34 we show the tuned model fit (in blue) provided by such a set of parameters, which wildly overfits the training data. In the top-

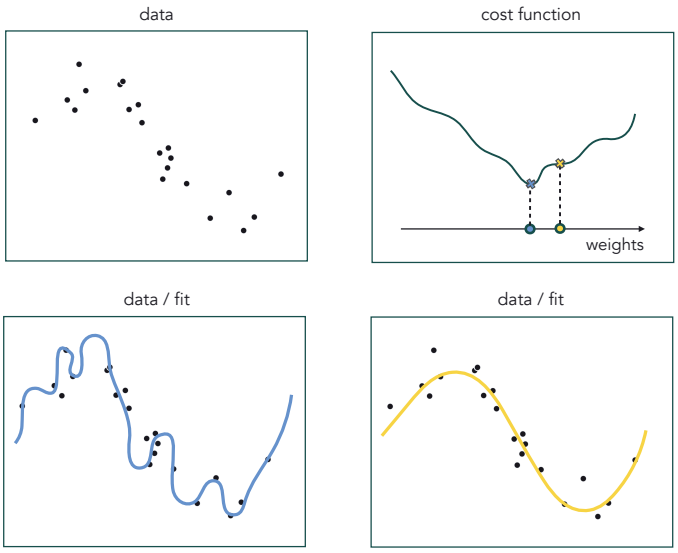

Figure 11.34 (top-left panel) A generic nonlinear regression dataset. (top-right panel) A figurative illustration of the cost function associated with a high-capacity model over the training portion of this data. The global minimum is marked here with a blue dot (along with its evaluation by a blue `x`) and a point nearby is marked in yellow (whose evaluation is shown as a yellow `x`). (bottom-left panel) The original data and fit (in blue) provided by the model using parameters from the global minimum of the cost function severely overfits the training portion of the data. (bottom-right panel) The parameters corresponding to the yellow dot shown in the top-right panel minimize the cost function over the validation portion of the data, and thus provide a much better fit (in yellow) to the data. See text for further details.

right panel we also show a set of parameters lying relatively near the global minimum as a yellow dot, and whose evaluation of the function is shown as a yellow `x`. This set of parameters lying in the general neighborhood of the global minimum is where the cost function is minimized over the *validation* portion of our data. Because of this the corresponding fit (shown in the bottom-right panel in yellow) provides a much better representation of the data.

This toy example is illustrative of a more general principle we have seen earlier in Section 11.3.2: that overfitting is due both to the *capacity* of an untuned model being too high *and* its corresponding cost function (over the training data) being *optimized* too well, leading to an overly complex tuned model. This phenomenon holds true for all machine learning problems (including regression, classification, and unsupervised learning techniques like the Autoencoder) and is the motivation for general regularization based cross-validation strategies: if proper optimization of *all parameters* of a high-capacity model leads to overfitting, it can be avoided by optimizing said model *imperfectly* when validation error (not training error) is at its lowest. In other words, *regularization* in the

context of cross-validation constitutes a set of approaches to cross-validation wherein we carefully tune all parameters of a high-capacity model by setting them purposefully away from the global minimum of its associated cost function. This can be done in a variety of ways, and we detail the two most popular approaches next.

11.6.2 Early stopping based regularization

With *early stopping* based regularization[12] we properly tune a high-capacity model by making a run of local optimization (tuning all parameters of the model), and by using the set of weights from this run where the model achieves minimum validation error. This idea is illustrated in the left panel of Figure 11.35 where we employ the same prototypical cost function first shown in the top-right panel of Figure 11.34. During a run of local optimization we frequently compute training and validation errors (e.g., at each step of the optimization procedure) so that a set of weights providing minimum validation error can be determined with fine resolution.

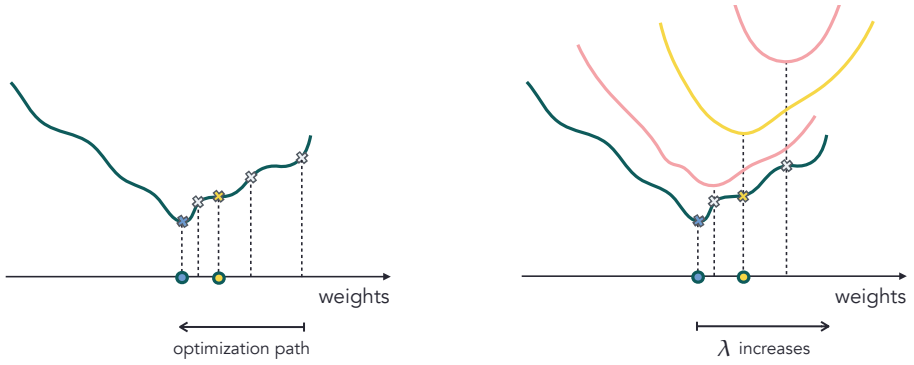

Figure 11.35 (left panel) A figurative illustration of early stopping regularization applied to a prototypical cost function of a high-capacity model. We make a run of optimization – here shown to completion at the global minimum in blue – and choose the set of weights that provide a minimum validation error (shown in yellow). (right panel) A figurative illustration of regularizer based regularization. By adding a regularizer function to the cost associated with a high-capacity model we change its shape, in particular dragging its global minimum (where overfitting behavior occurs) away from its original location. The regularized cost function can then be *completely* minimized to recover weights as close to/far away from the true global minimum of the original cost function, depending on the choice of the regularization parameter λ. Proper setting of this parameter allows for the recovery of validation-error-minimizing weights. See text for further details.

Whether one literally stops the optimization run when minimum validation

[12] This regularization approach is especially popular when employing deep neural network models as detailed in Section 13.7.

error has been reached (which can be challenging in practice given the somewhat unpredictable behavior of validation error as first noted in Section 11.4.2) or one runs the optimization to completion (picking the best set of weights afterwards), in either case we refer to this method as early stopping regularization. Note that the method itself is analogous to the early stopping procedure outlined for boosting based cross-validation in Section 11.5.3, in that we sequentially increase the complexity of a model until minimum validation is reached. However, here (unlike boosting) we do this by controlling how well we optimize a model's parameters *simultaneously*, as opposed to one unit at a time.

Supposing that we begin our optimization with a small initial value (which we typically do) the corresponding training and validation error curves will, in general,[13] look like those shown in the top panel of Figure 11.36. At the start of the run the complexity of our model (evaluated at the initial weights) is quite small, providing a large training and validation error. As minimization proceeds, and we continue optimizing one step at a time, error in both training and validation portions of the data decreases while the complexity of the tuned model increases. This trend continues up until a point when the model complexity becomes too great and overfitting begins, and validation error increases.

In terms of the capacity/optimization dial conceptualization detailed in Section 11.3.2, we can think of (early stopping based) regularization as beginning with our capacity dial set all the way to the *right* (since we employ a high-capacity model) and our optimization dial all the way to the *left* (at the initialization of our optimization). With this configuration – summarized visually in the bottom panel of Figure 11.36 – we allow our optimization dial to directly govern the amount of complexity our tuned models can take. In other words, with this configuration our optimization dial becomes (roughly speaking) the ideal complexity dial described at the start of the chapter in Section 11.1.3. With early stopping we turn our optimization dial from left to right, starting at our initialization making a run of local optimization one step at a time, seeking out a set of parameters that provide minimum validation error for our (high-capacity) model.

There are a number of important engineering details associated with implementing an effective early stopping regularization procedure, which we discuss below.

- **Different optimization runs may lead to different tuned models.** The cost function topology associated with high-capacity models can be quite complicated. Different initializations can thus produce different trajectories towards potentially different minima of the cost function, and produce corresponding validation-error-minimizing models that differ in shape – as illustrated pictorially in the top row of Figure 11.37. However, in practice these differences

[13] Note that both can oscillate in practice depending on the optimization method used.

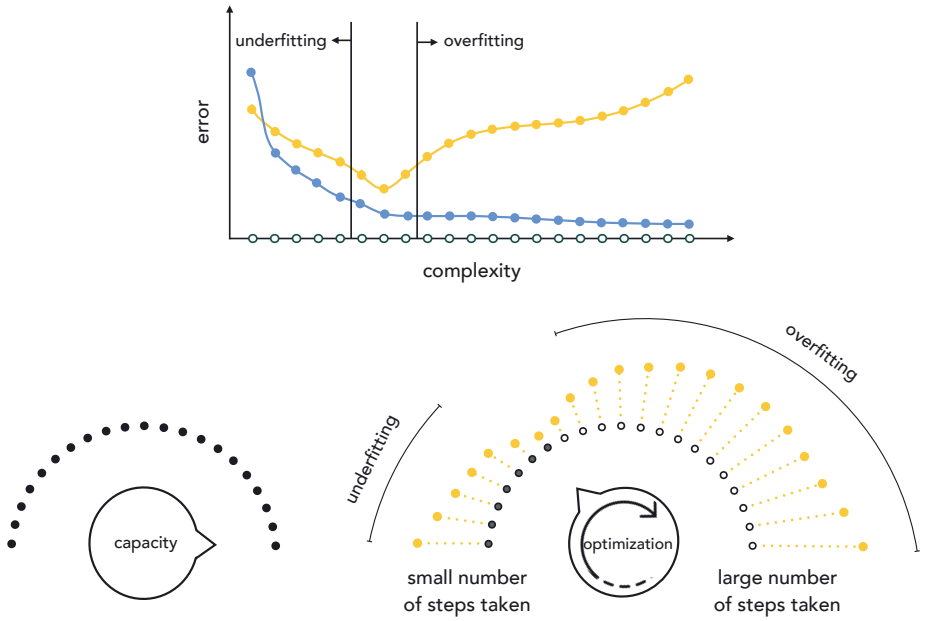

Figure 11.36 (top panel) A prototypical pair of training/validation error curves associated with a generic run of early stopping regularization. (bottom panels) With early stopping we set our capacity dial all the way to the *right* and our optimization dial all the way to the *left*. We then slowly move our optimization dial from left to right, iteratively improving the fit of our of model to the training data, adjusting all of its parameters simultaneously one step at a time. As each step of optimization progresses we slowly turn the optimization dial clockwise from left to right, gradually increasing the complexity of our tuned model, in search of a tuned model with minimum validation error. Here each notch on the optimization dial abstractly denotes a step of local optimization. See text for further details.

tend not to effect performance, and the resulting models can be easily combined or *bagged* together (see Section 11.9) to average out any major differences in their individual performance.

- **How high should capacity be set?** How do we know how high to set the capacity of our model when using early stopping (or any other form of) regularization based cross-validation? In general there is no single answer. It must simply be set at least "high" enough that the model overfits if optimized completely. This can be achieved by adjusting M (the number of units in the model) and/or the capacity of individual units (by, for example, using shallow versus deep neural network or tree based units, as we detail in Chapters 13 and 14, respectively).

- **Local optimization must be carefully performed.** One must be careful with the sort of local optimization scheme used with early stopping cross-validation.

As illustrated in the bottom-left panel of Figure 11.37, ideally we want to turn our optimization dial smoothly from left to right, searching over a set of model complexities with a fine resolution. This means, for example, that with early stopping we often avoid local optimization schemes that take very large steps (e.g., Newton's method – as detailed in Chapter 4) as this can result in a coarse and low-resolution search over model complexity that can easily skip over minimum-validation models, as depicted in the bottom-right panel of the figure. Local optimizers that take smaller, high-quality steps – like the advanced first-order methods detailed in Appendix A – are often preferred when employing early stopping. Moreover, when employing mini-batch/stochastic first-order methods (see Appendix Section A.5) validation error should be measured *several times per epoch* to avoid taking too many steps without measuring validation error.

- **When is validation error really at its lowest?** While generally speaking validation error decreases at the start of an optimization run and eventually increases (making somewhat of a "U" shape) it can certainly fluctuate up and down during optimization. Therefore it is not all together obvious when the validation error has indeed reached its lowest point unless the optimization process is performed to completion. To deal with this peculiarity, often in practice a reasonable engineering choice is made as to when to stop based on how long it has been since the validation error has *not* decreased. Moreover, as mentioned earlier, one need not truly halt a local optimization procedure to employ the thrust of early stopping, and can simply run the optimizer to completion and select the best set of weights from the run after completion.

The interested reader can see Example 13.14 for a simple illustration of early stopping based regularization.

11.6.3 Regularizer based methods

A *regularizer* is a simple function that can be added to a machine learning cost for a variety of purposes, e.g., to prevent unstable learning (as we saw in Section 6.4.6), as a natural part of relaxing the Support Vector Machine (Section 6.5.4) and multi-class learning scenarios (Section 7.3.4), and for feature selection (Section 9.7). As we will see, the latter of these applications (feature selection) is very similar to our use of the regularizer here.

Adding a simple regularizer function like one of those we have seen in previous applications (e.g., the ℓ_2 norm) to the cost of a high-capacity model, we can alter its shape and, in particular, move the location of its global minimum away from its original location. In general if our high-capacity model is given as model $(\mathbf{x}, \Theta)$, its associated cost function given by g, and a regularizer h, then the regularized cost is given as the linear combination of g and h as

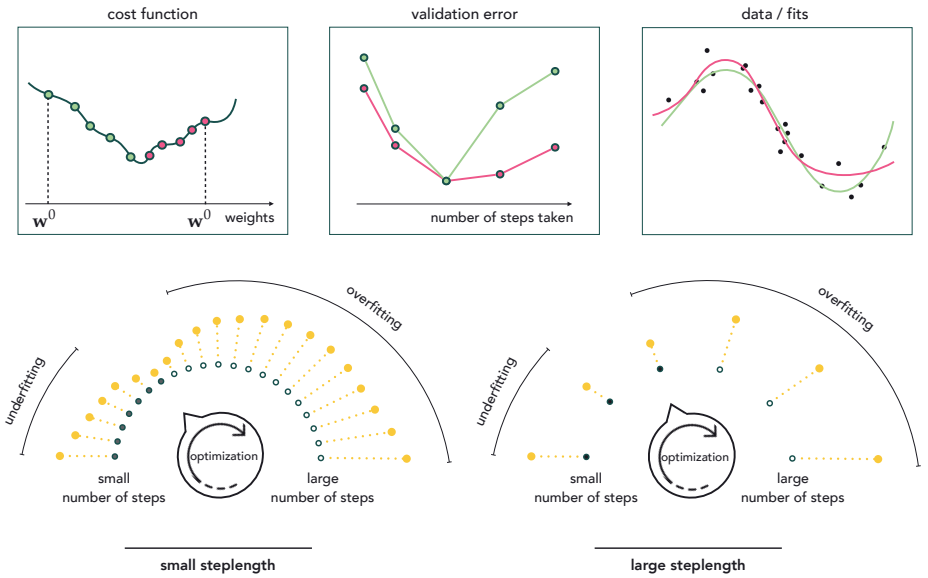

Figure 11.37 Two subtleties associated with early stopping based regularization. (top-left panel) A prototypical cost function associated with a high-capacity model, with two optimization paths (shown in red and green, respectively) resulting from two local optimization runs beginning at different starting points. (top-middle panel) The validation error histories corresponding to each optimization run. (top-right panel) While each run produces a different set of optimal weights, and a different fit to the data (here shown in green and red respectively, corresponding to each run), these fits are generally equally representative. (bottom-left panel) Taking optimization steps with a small steplength makes the early stopping procedure a fine-resolution search for optimal model complexity. With such small steps we smoothly turn the optimization dial from left to right in search of a validation-error-minimizing model. (bottom-right panel) Using steps with a large steplength makes early stopping a coarse-resolution search for optimal model complexity. With each step taken we aggressively turn the dial from left to right, performing a coarser model search that potentially skips over the optimal model.

$$g(\Theta) + \lambda h(\Theta) \tag{11.38}$$

where λ is referred to as the *regularization parameter*. The regularization parameter is always nonnegative $\lambda \geq 0$ and controls the mixture of the cost and regularizer. When it is set small and close to zero the regularized cost is essentially just g, and conversely when set very large the regularizer h dominates in the linear combination (and so upon minimization we are really just minimizing it alone). In the right panel of Figure 11.35 we show how the shape of a figurative regularized cost (and consequently the location of its global minimum) changes with the value of λ.

Supposing that we begin with a large value of λ and try progressively smaller

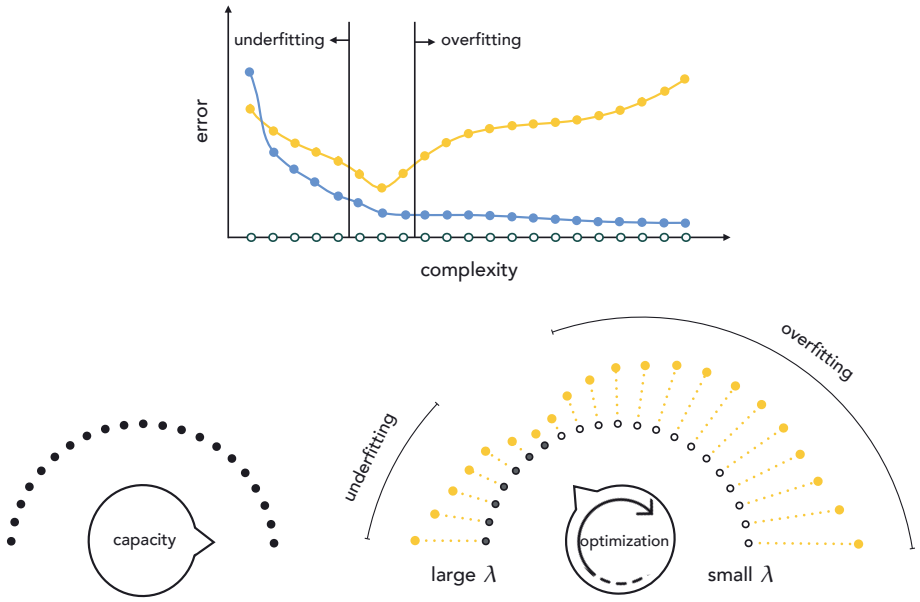

Figure 11.38 (top panel) A prototypical pair of training/validation error curves associated with a generic run of regularizer based cross-validation. (bottom panels) With regularizer-based cross-validation we set our capacity dial all the way to the *right* and our optimization dial all the way to the *left* (beginning with a large value for our regularization parameter λ). We then slowly move our optimization dial from left to right by *decreasing* the value of λ, where here each notch on the optimization dial represents the complete minimization of the corresponding regularized cost function in Equation (11.38), improving the fit our of model to the training data. By adjusting the value of λ (and completely minimizing each corresponding regularized cost) we slowly turn the optimization dial clockwise from left to right, gradually increasing the complexity of our tuned model, in search of a model with minimum validation error. See text for further details.

values (completely optimizing each regularized cost), the corresponding training and validation error curves will in general look something like those shown in the top panel of Figure 11.38 (remember in practice that *validation error* can oscillate, and need not take just one dip down). At the start of this procedure, using a large value of λ, the complexity of our model is quite small as the regularizer completely dominated in the regularized cost, and thus the associated minimum recovered belongs to the regularizer and not the cost function itself. Since the set of weights is virtually unrelated to the data we are training over, the corresponding model will tend to have large training and validation errors. As λ is decreased the parameters provided by complete minimization of the regularized cost will be closer to the global minimum of the original cost itself, and so error on both training and validation portions of the data decreases while (generally speaking) the complexity of the tuned model increases. This trend continues up until a point when the regularization parameter is small enough

that the recovered parameters lie too close to that of the original cost, so that the corresponding model complexity becomes too great. Here overfitting begins and validation error increases.

In terms of the capacity/optimization dial scheme detailed in Section 11.3.2, we can think of regularizer based cross-validation as beginning with our capacity dial set to the *right* (since we employ a high-capacity model) and our optimization dial all the way to the *left* (employing a large value for λ in our regularized cost). With this configuration (summarized visually in the bottom panel of Figure 11.38) we allow our optimization dial to directly govern the amount of complexity our tuned models can take. As we turn our optimization dial from left to right we *decrease* the value of λ and *completely* minimize the corresponding regularized cost, seeking out a set of parameters that provide minimum validation error for our (high-capacity) model.

There are a number of important engineering details associated with implementing an effective regularizer based cross-validation procedure, which we discuss below.

- **Bias weights are often not included in the regularizer.** As with linear models as discussed in Section 9.7, often only the nonbias weights of a general model are included in the regularizer. For example, suppose that we employ fixed-shape universal approximator units and hence our parameter set Θ contains a single bias w_0 and feature-touching weights w_1, w_2, ..., w_B. If we then regularize our cost function $g(\Theta)$ using the squared ℓ_2 norm, our regularized cost would then take the form $g(\Theta) + \lambda \sum_{b=1}^{B} w_b^2$. When employing neural network units we follow the same pattern, but here we have far more bias terms to avoid including in the regularizer. For example, if we use units of the form $f_b(\mathbf{x}) = \tanh\left(w_{b,0} + x_1 w_{b,1} + \cdots + x_N w_{b,N}\right)$ the term $w_{b,0}$ – internal to the unit – is a bias term we also do not want included in our regularizer. Thus, to regularize a cost function including these units using the squared ℓ_2 norm we have $g(\Theta) + \lambda \left(\sum_{b=1}^{B} w_b^2 + \sum_{b=1}^{B} \sum_{n=1}^{N} w_{b,n}^2\right)$.

- **Choice of regularizer function.** Note that while the ℓ_2 norm is a very popular regularizer, one can – in principle – use any simple function as a regularizer. Other popular choices of regularizer functions include the ℓ_1 norm regularizer $h(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_{n=1}^{N} |w_n|$, which tends to produce *sparse* weights, and the total variation regularizer $h(\mathbf{w}) = \sum_{n=1}^{N-1} |w_{n+1} - w_n|$, which tends to produce *smoothly-varying* weights. We often use the simple quadratic regularizer (ℓ_2 norm squared) to incentivize weights to be *small*, as we naturally do with two-class and multi-class logistic regression. Each of these different kinds of regularizers tends to pull the global minimum of the sum towards different portions of the input space – as illustrated in Figure 11.39 for the quadratic (top-left panel), ℓ_1 norm (top-middle panel), and total variation norm (top-right panel).

- **Choosing the range of λ values.** Analogously to what we saw with early stopping and boosting procedures previously, with regularization we want to perform our search as carefully as possible, turning our optimization dial as smoothly as possible from left to right in search of our perfect model. This desire translates directly to both the range and number of values for λ that we test out. For instance, the more values we try within a given range, the smoother we turn our optimization dial (as depicted visually in the bottom-left panel of Figure 11.39). The limit on how many values we can try is often dictated by computation and time restrictions, since for *every* value of λ tried a complete minimization of a corresponding regularized cost function must be performed. This can make regularizer based cross-validation very computationally expensive. On the other hand, trying too few of values can result in a coarse search for weights providing minimum validation error, increasing the possibility that such weights are skipped over entirely (as depicted in the bottom-right panel of Figure 11.39).

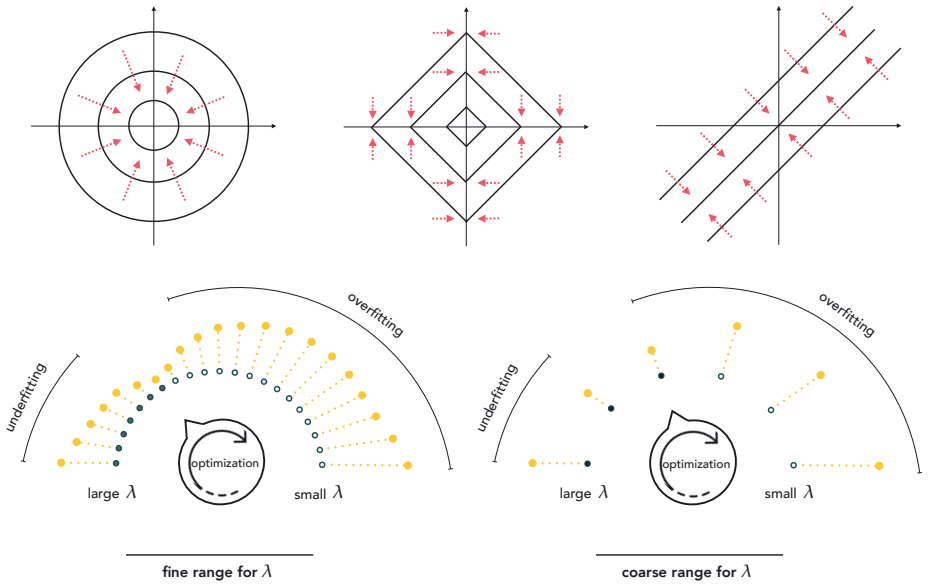

Figure 11.39 (top row) A visual depiction of where the ℓ_2 (top-left panel), ℓ_1 (top-middle panel), and total variation (top-right panel) functions pull the global minimum of a cost function – when used as a regularizer. These functions pull the global minimum towards the origin, the coordinate axes, and diagonal lines where consecutive entries are equal, respectively. (bottom-left panel) Testing out a large range and number of values for the regularization parameter λ results in a fine-resolution search for validation-error-minimizing weights. (bottom-right panel) A smaller number (or a poorly chosen range) of values can result in a coarse search that can skip over ideal weights. See text for further details.

Example 11.13 Tuning λ for a two-class classification problem
In this example we use a quadratic regularizer to find a proper nonlinear classifier for the two-class classification dataset shown in the left column of Figure 11.40 where the training set is shown with their perimeter colored in light blue, and the validation points have their perimeter colored yellow. Here we use $B = 20$ neural network units – a high-capacity model with respect to this data – and try out 6 values of λ uniformly distributed between 0 and 0.5 (completely minimizing the corresponding regularized cost in each instance). As the value of λ changes the fit provided by the weights recovered from the global minimum of each regularized cost function is shown in the left column, while the corresponding training and validation errors are shown in blue and yellow, respectively, in the right column. In this simple experiment, a value somewhere around $\lambda \approx 0.25$ appears to provide the lowest validation error and corresponding best fit to the dataset overall.

11.6.4 Similarity to regularization for feature selection

Akin to the boosting procedure detailed in the previous section, here the careful reader will notice how similar the regularizer based framework described here is to the concept of regularization detailed for feature selection in Section 9.7. The two approaches are very similar in theme, except here we do not select from a set of given input features but *create* them ourselves based on a universal approximator. Additionally, instead of our main concern with regularization being *human interpretability* of a machine learning model, as it was in Section 9.7, here we use regularization as a tool for cross-validation.

11.7 Testing Data

In Section 11.3.4 we saw how, in place of training error, *validation error* is an appropriate measurement tool that enables us to accurately identify an appropriate model/parameter tuning for generic data. However, like the training error, choosing a model based on minimum validation error can also potentially lead to models that *overfit* our original dataset. In other words, at least in principle, we can overfit to validation data as well. This can make validation error a poor indicator of how well a cross-validated model will perform in general. As we will see in this brief section, the potential dangers of this reality can be ameliorated, provided the dataset is large enough, by splitting our original training data into not two sets (training and validation), but *three*: training, validation, and testing sets. By measuring a cross-validated model's performance on the *testing set* we not only gain a better measure of its ability to capture the true nature of the

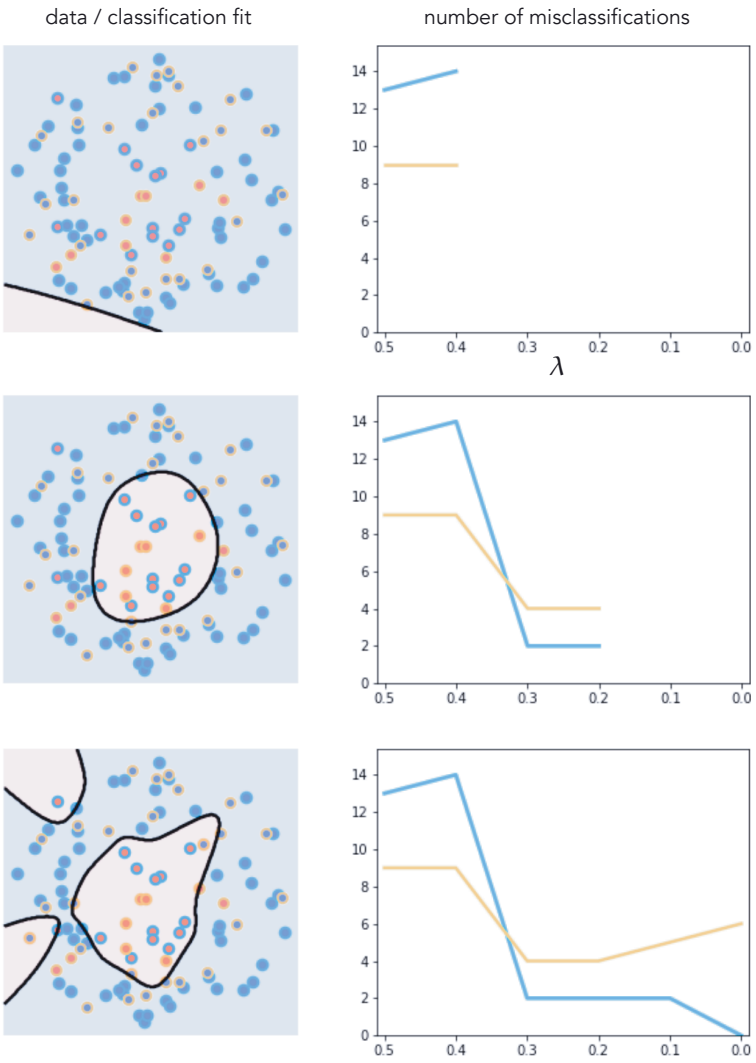

Figure 11.40 Figure associated with Example 11.13. See text for details.

phenomenon generating the data, but we also gain a reliable measurement tool for comparing the efficacy of multiple cross-validated models.

11.7.1 Overfitting validation data

In Section 11.3.4 we learned how, as a measurement tool, training error fails to help us properly identify when a tuned model has sufficient complexity to properly represent a given dataset. There we saw how an overfitting model, that is one that achieves minimum training error but is far too complex, represents the

training data we currently have incredibly well, but simultaneously represents the phenomenon underlying the data (as well as any future data similarly generated by it) very poorly. While not nearly as prevalent in practice, it is possible for a properly cross-validated model to *overfit validation data*.

To see how this is possible let us analyze an extreme two-class classification dataset. As shown in the left panel of Figure 11.41 this dataset shows no meaningful relationship whatsoever between the input and output (labels). Indeed we created it by choosing the coordinates of the two-dimensional input points randomly over the input space, and then assigning label value +1 (red class) to half of the points (which are selected, once again, at random) and label value −1 (blue class) to the other half.

and points randomly (uniformly) on the unit square and assigned labels to the points at random.

Because we know that the underlying phenomenon generating this dataset is *completely random*, no model, whether it has been found via cross-validation or otherwise, should ever allow us to correctly predict the label of future points with an accuracy that is substantially greater than 50 percent. In other words, no model should truly provide better-than-chance accuracy on random data such as this. However, this reality need not be reflected in an appropriately cross-validated model (i.e., one with minimum validation error for some split of the data). Indeed in the right panel of Figure 11.41 we show the decision boundary of a naively cross-validated model for this dataset, where $\frac{1}{5}$ of the original data was used as validation, and color the regions according to the model's predictions. This particular cross-validated model provides 70 percent accuracy on the validation data, which perhaps at first glance is mysterious given our understanding of the underlying phenomenon. However, this is because, even though it was chosen as the validation-error-minimizing model, this model still *overfits* the original data. While it is not as prevalent or severe as the overfitting that occurs with training-error-minimized models, overfitting to validation data like this is still a danger that in practice should be avoided when possible.

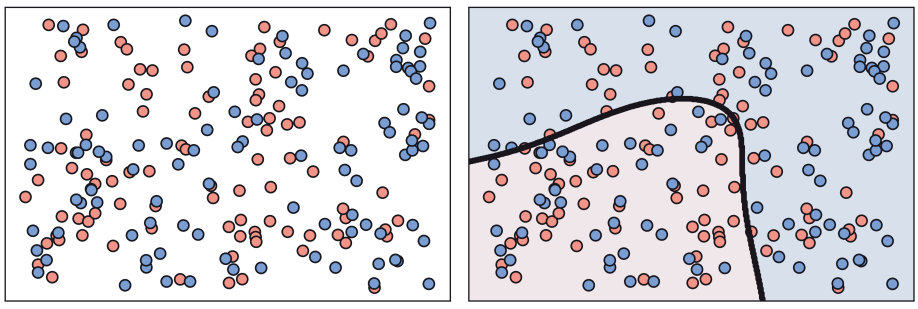

Figure 11.41 (left panel) A randomly generated two-class classification dataset. (right panel) The decision boundary of a cross-validated model providing 70 percent accuracy on the validation data, which is meaningfully greater than random chance (50 percent).

11.7.2 Testing data and testing error

Up until now we have used validation data *both* to select the best model for our data (i.e., cross-validation) *and* to measure its quality. However, much like when the notion of validation data was first introduced in Section 11.3.4, we see that using the same set of data to perform both of these tasks can lead to the selection of an overfitting model and can diminish the utility of validation error as a measure of model quality. The solution to this problem (again much like when we introduced the concept of validation data to begin with) is to split up the two tasks we now assign to validation data by introducing a *second* validation set. This "second validation set" is often called a *test set* or *testing set*, and is used solely to measure the quality of our final cross-validated model.

By splitting our data into three chunks (as illustrated in Figure 11.42) we still use training and validation portions precisely as they have been used thus far (i.e., for performing cross-validation). However, after the cross-validated model is constructed its quality is measured on the distinct *testing set* of data, on which it has been neither trained nor validated. This *testing error* gives an "unbiased" estimate of the cross-validated model's performance, and is generally closer to capturing the true error of our model on future data generated by the same phenomenon.

Figure 11.42 The original dataset (left panel) is split randomly into three nonoverlapping subsets: training, validation, and testing sets (right panel).

In the case of our random two-class data introduced in Section 11.7.1, such a testing set provides a far more accurate picture of how well our cross-validated model will work in general. In Figure 11.43 we again show this dataset (with validation data points highlighted with yellow boundaries), which is now augmented by the addition of a testing portion (those points highlighted with green boundaries) that are generated precisely the same way we created the original dataset in Figure 11.41. Note importantly that this testing portion was not used during training/cross-validation. While our cross-validated model achieved 70 percent accuracy on the validation set (as mentioned previously), it achieves only a 52 percent accuracy on the testing set, which is a more realistic indicator of our model's true classification ability, given the nature of this data.

What portion of the original dataset should we assign to our testing set? As with the portioning of training and validation (detailed in Section 11.3.4), there is no general rule here, save perhaps one: the use of testing data is a luxury we

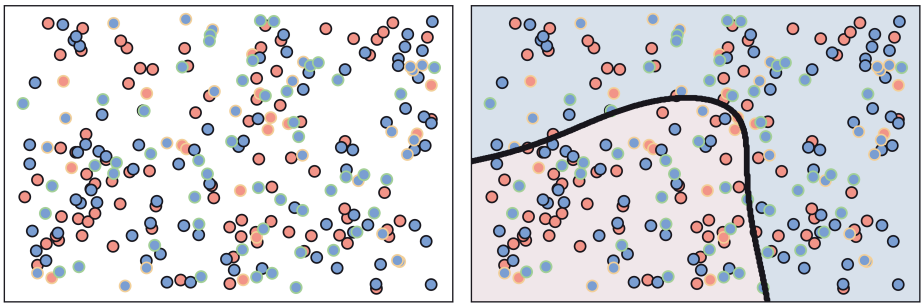

Figure 11.43 (left panel) The same dataset first shown in Figure 11.41 augmented by data points highlighted in green that were removed from training and validation procedures, and left out for testing. (right panel) The cross-validated model only achieves a 52 percent accuracy on the testing set, which is a much better estimate of a machine learning model's ability to classify random data.

can indulge in only when we have a large amount of data. When data is scarce we must leverage it all just to build a "halfway reasonable" cross-validated model. When data is plentiful, however, often the size of validation and testing sets are chosen similarly. For example, if $\frac{1}{5}$ of a dataset is used for validation, often for simplicity the same portion is used for testing as well.

11.8 Which Universal Approximator Works Best in Practice?

Beyond special circumstances such as those briefly discussed below, it is virtually never clear *a priori* which, if any, of the universal approximators will work best. Indeed cross-validation (as outlined in the previous sections) is *the* toolset one uses in practice to decide which type of universal approximator based model works best for a particular problem. Using these techniques one can create a range of different cross-validated models, each built from a distinct type of universal approximator, and compare their efficacy on a testing set (described in Section 11.7) afterwards to see which universal approximator works best. Alternatively, one can cross-validate a range of universal approximator based models and average them together afterwards, as discussed next in Section 11.9, leading to an averaged model that consists of representatives from multiple universal approximators.

In some instances broad understanding of a dataset can direct the choice of universal approximator. For example, because oftentimes business, census, and (more generally) *structured* datasets consist of broad mixtures of continuous and discontinuous categorical input features (see Section 6.7.1), tree-based universal approximators, with their discontinuous step-like shapes, often provide stronger results on average than other universal approximator types. On the other hand, data that is naturally continuous (e.g., data generated by natural processes or

sensor data) is often better matched with a continuous universal approximator: fixed-shape or neural network. Understanding whether future predictions need be made *inside* or *outside* the input domain of the original dataset can also help guide the choice of approximator. In such cases fixed-shape or neural network approximators can be preferred over trees – the latter by their very nature always creating perfectly flat predictions outside of the original data's input domain (see Exercise 14.9 for further details).

When *human interpretability* is of primary importance, this desire (in certain circumstances) can drive the choice of universal approximator. For example, due to their discrete branching structure (see Section 14.2), tree-based universal approximators can often be much easier to interpret than other approximators (particularly neural networks). For analogous reasons fixed-shape approximators (e.g., polynomials) are often employed in the natural sciences, like the gravitational phenomenon underlying the Galileo's ramp dataset discussed in Example 11.17.

11.9 Bagging Cross-Validated Models

As we discussed in detail in Section 11.3.4, validation data is the portion of our original dataset we exclude at random from the training process in order to determine a proper tuned model that will faithfully represent the phenomenon generating our data. The validation error generated by our tuned model on this "unseen" portion of data is the fundamental measurement tool we use to determine an appropriate cross-validated model for our entire dataset (besides, perhaps, a testing set – see Section 11.7). However, the random nature of splitting data into training and validation poses an obvious flaw to our cross-validation process: what if the random splitting creates training and validation portions which are not desirable representatives of the underlying phenomenon that generated them? In other words, in practice what do we do about potentially bad training–validation splits, which can result in poorly representative cross-validated models?

Because we *need* cross-validation in order to choose appropriate models in general, and because we can do nothing about the (random) *nature* by which we split our data for cross-validation (what better method is there to simulate the "future" of our phenomenon?), the practical solution to this fundamental problem is to simply create several different training–validation splits, determine an appropriate cross-validated model on each split, and then *average* the resulting cross-validated models. By averaging a set of cross-validated models, also referred to as *bagging* in the jargon of machine learning, we can very often "average out" the potentially undesirable characteristics of each model while synergizing their positive attributes. Moreover, with *bagging* we can also effectively combine cross-validated models built from *different* universal approximators. Indeed this

is the most reasonable way of creating a single model built from different types of universal approximators in practice.

Here we will walk through the concept of bagging or model averaging for regression, as well as two-class and multi-class classification by exploring an array of simple examples. With these simple examples we will illustrate the superior performance of bagged models visually, but in general we confirm this using the notion of testing error (see Section 11.7) or an estimate of testing error (often employed when bagging trees – see Section 14.6). Regardless, the principles detailed here can be employed more widely as well to any machine learning problem. As we will see, the best way to average/bag a set of cross-validated regression models is by taking their *median* and cross-validated classification models by computing the *mode* of their predicted labels.

11.9.1 Bagging regression models

Here we explore several ways of bagging a set of cross-validated models for the nonlinear regression dataset first described in Example 11.6. As we will see, more often than not the best way to bag (or average) cross-validated regression models is by taking their *median* (as opposed to their *mean*).

Example 11.14 Bagging cross-validated regression models

In the set of small panels in the left side of Figure 11.44 we show ten different training–validation splits of a prototypical nonlinear regression dataset, where $\frac{4}{5}$ of the data in each instance has been used for training (colored light blue) and $\frac{1}{5}$ is used for validation (colored yellow). Plotted with each split of the original data is the corresponding cross-validated model found via naive cross-validation (see Section 11.4.2) of the full range of polynomial models of degree 1 to 20. As we can see, while *many* of these cross-validated models perform quite well, several of them (due to the particular training–validation split on which they are based) severely *underfit* or *overfit* the original dataset. In each instance the poor performance is completely due to the particular underlying (random) training–validation split, which leads cross-validation to a validation-error-minimizing tuned model that still does not represent the true underlying phenomenon very well. By taking an *average* (here the *median*) of the ten cross-validated models shown in these small panels we can average out the poor performance of this handful of bad models, leading to a final bagged model that fits the data quite well – as shown in the large right panel of Figure 11.44.

Why average our cross-validated models using the *median* as opposed to the *mean*? Simply because the mean is far more sensitive to *outliers* than is the median. In the top row of Figure 11.45 we show the regression dataset shown previously along with the individual cross-validated fits (left panel), the median bagged model (middle panel), and the mean bagged model (right panel). Here the mean model is highly affected by the few overfitting models in the group,

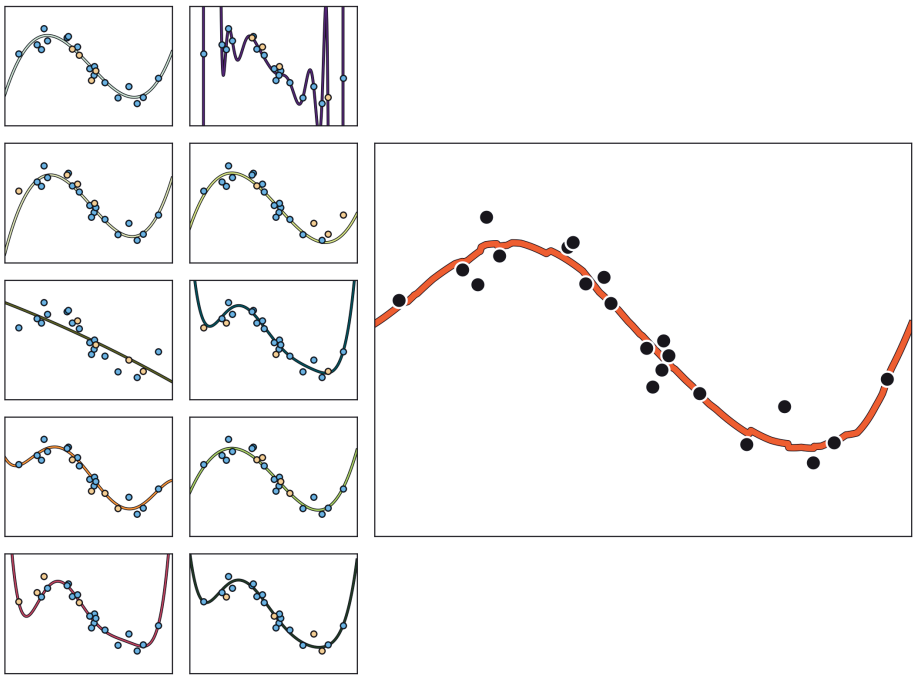

Figure 11.44 Figure associated with Example 11.14. (left columns) The results of applying cross-validation to ten random training–validation splits of a nonlinear regression dataset, with each resulting model shown in one of the ten small panels. Here the training and validation portions in each instance are colored light blue and yellow, respectively. (right column) The fit, shown in red, resulting from the bagging of the ten models whose fits are shown on the left. See text for further details.

and ends up being far too oscillatory to fairly represent the phenomenon underlying the data. The median is not affected in this way, and is therefore a much better representative.

When we bag we are simply averaging various cross-validated models with the desire to both avoid bad aspects of poorly-performing models, and jointly leverage strong elements of the well-performing ones. Nothing in this notion prevents us from bagging together cross-validated models built using different universal approximators, and indeed this is the most organized way of combining different types of universal approximators in practice.

In the bottom row of Figure 11.45 we show the result of a cross-validated polynomial model (left panel), a cross-validated neural network model (second to the left panel), and a cross-validated tree-based model (second to the right panel) built via boosting (see Section 11.5). Each cross-validated model uses a different training–validation split of the original dataset, and the bagged median of these models is shown in the right panel.

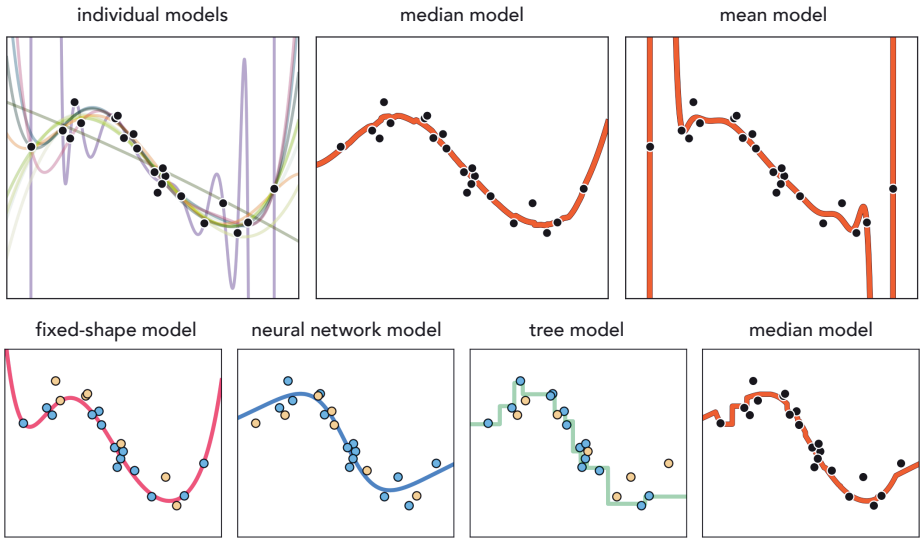

Figure 11.45 Figure associated with Example 11.14. The ten individual cross-validated models first shown in Figure 11.45 are plotted together in the top-left panel. The *median* and *mean* of these models are shown in the top-middle and top-right panel, respectively. With regression, bagging via the median tends to produce better results as it is less sensitive to outliers. (bottom row) Cross-validated fixed-shape polynomial (left panel), neural network (second panel from the left), and tree-based (second panel from the right) models. The median of these three models is shown in the right panel. See text for further details.

11.9.2 Bagging classification models

The principle behind bagging cross-validated models holds analogously for classification tasks, just as it does with regression. Because we cannot be certain whether or not a particular (randomly chosen) validation set accurately represents the "future data" from a given phenomenon well, the averaging (or bagging) of a number of cross-validated classification models provides a way of averaging out poorly representative portions of some models while combining the various models' positive characteristics.

Because the predicted output of a (cross-validated) classification model is a *discrete* label, the average used to bag cross-validated classification models is the *mode* (i.e., the most popularly predicted label).

Example 11.15 Bagging cross-validated two-class classification models

In the set of small panels in the left column of Figure 11.46 we show five different training–validation splits of the prototypical two-class classification dataset first described in Example 11.7, where $\frac{2}{3}$ of the data in each instance is used for training and $\frac{1}{3}$ is used for validation (the boundaries of these points

are colored yellow). Plotted with each split of the original data is the nonlinear decision boundary corresponding to each cross-validated model found via naive cross-validation of the full range of polynomial models of degree 1 to 8. Many of these cross-validated models perform quite well, but some of them (due to the particular training–validation split on which they are based) severely *overfit* the original dataset. By bagging these models using the most popular prediction to assign labels (i.e., the *mode* of these cross-validated model predictions) we produce an appropriate decision boundary for the data shown in the right panel of the figure.

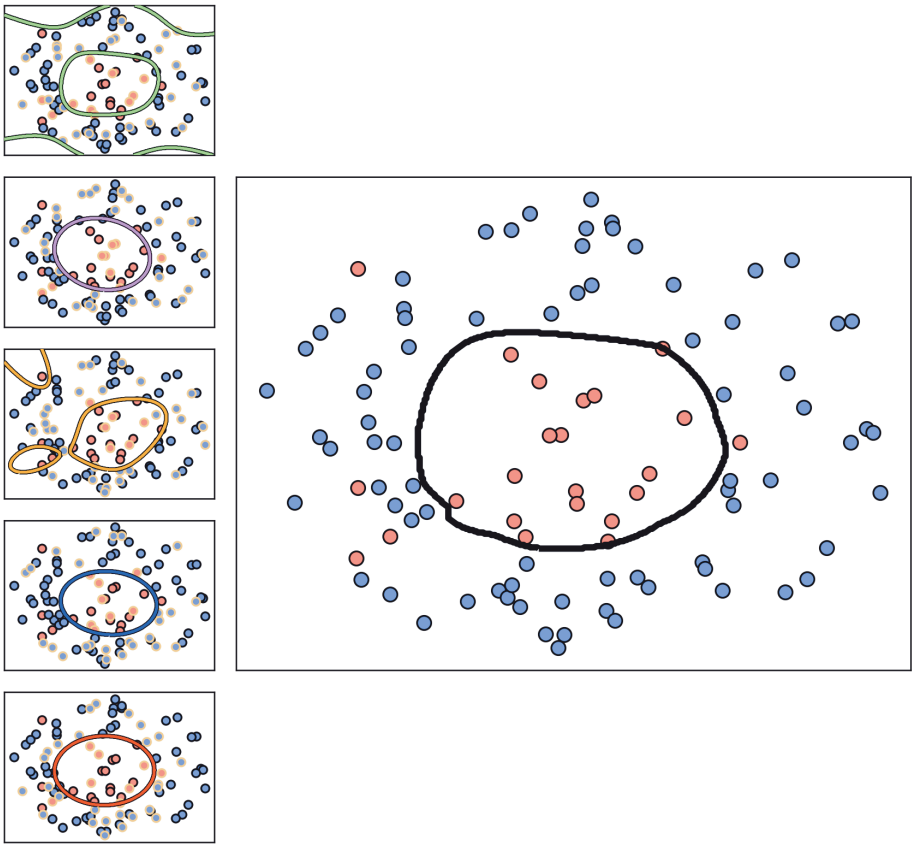

Figure 11.46 Figure associated with Example 11.15. (left column) Five models cross-validated on random training–validation splits of the data, with the validation data in each instance highlighted with a yellow outline. The corresponding nonlinear decision boundary provided by each model is shown in each panel. Some models, due to the split of the data on which they were built, severely overfit. (right column) The original dataset with the decision boundary provided by the bag (i.e., mode) of the five cross-validated models. See text for further details.

In the top-middle panel of Figure 11.47 we illustrate the decision boundaries of

five cross-validated models, each built using $B = 20$ neural network units trained on different training–validation splits of the dataset shown in the top-left panel of the figure. In each instance $\frac{1}{3}$ of the dataset is randomly chosen as validation (highlighted in yellow). While some of the learned decision boundaries (shown in the top-middle panel) separate the two classes quite well, others do a poorer job. In the top-right panel we show the decision boundary of the bag, created by taking the mode of the predictions from these cross-validated models, which performs quite well.

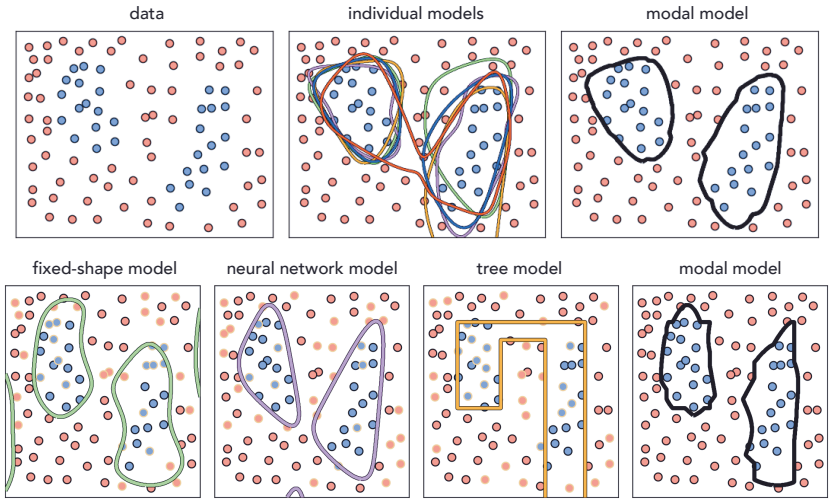

Figure 11.47 Figure associated with Example 11.15. (top-left panel) A toy two-class classification dataset first described in Example 11.7. (top-middle panel) The decision boundaries, each shown in a different color, resulting from five models cross-validated on different training-validation splits of the data. (top-right panel) The decision boundary resulting from the mode (the *modal model*) of the five individual cross-validated models. (bottom row) The decision boundaries provided by a cross-validated fixed-shape polynomial model (left panel), neural network model (second from the left panel), and tree-based model (third from the left panel). In each instance the validation portion of the data is highlighted in yellow. (right panel) The decision boundary provided by the mode of these three models. See text for further details.

As with regression, with classification we can also combine cross-validated models built from different universal approximators. We illustrate this in the bottom row of Figure 11.47 using the same dataset. In particular, we show the result of a cross-validated polynomial model (left panel), a cross-validated neural network model (in the second to the left panel), and a cross-validated tree-based model (second to the right panel). Each cross-validated model uses a different training–validation split of the original data, and the bag (mode) of these models shown in the right panel performs quite well.

Example 11.16 Bagging cross-validated multi-class classification models

In this example we illustrate the bagging of various cross-validated multi-class models on the two different datasets shown in the left column of Figure 11.48. In each case we naively cross-validate polynomial models of degree 1 through 5, with five cross-validated models learned in total. In the middle column of the figure we show the decision boundaries provided by each cross-validated model in distinct colors, while the decision boundary of the final *modal model* is shown in the right column for each dataset.

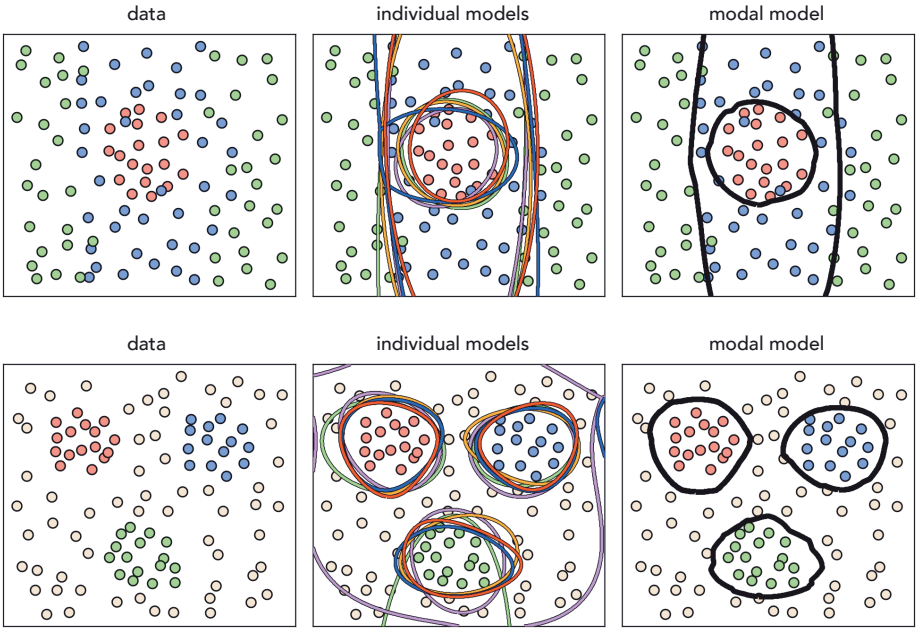

Figure 11.48 Figure associated with Example 11.16. (left column) Two multi-class classification datasets. (middle column) The decision boundaries resulting from five cross-validated models, each shown as a distinct color. (right column) The decision boundary provided by bagging. See text for further details.

11.9.3 How many models should we bag in practice?

Note that in the examples of this section the exact number of cross-validated models bagged were set somewhat arbitrarily. Like other important parameters involved with cross-validation (e.g., the portion of a dataset to reserve for validation) there is no magic number (of cross-validated models) used generally in practice for bagging. Ideally, if we knew that any random validation portion of a dataset generally represented it well, which is often true with very large datasets, there would be less of a need to ensemble multiple cross-validated

models where each was trained on a different training–validation split of the original data. Indeed in such instances we could instead bag a range of models trained on a single training–validation split in order to achieve similar improvements over a single model. On the other hand, the less we could trust in the faithfulness of a random validation portion to represents a phenomenon at large, the less we could trust an individual cross-validated model, and hence we might wish to bag more of them to help average our poorly-performing models resulting from bad splits of the data. Often in practice considerations such as computation power and dataset size determine if bagging is used, and if so, how many models are employed in the average.

11.9.4 Ensembling: Bagging versus Boosting

The bagging technique described here wherein we combine a number of different models, each cross-validated independently of the others, is a primary example of what is referred to as *ensembling* in the jargon of machine learning. An *ensembling method* (as the name "ensemble" implies) generally refers to any method of combining different models in a machine learning context. Bagging certainly falls into this general category, as does the general *boosting* approach to cross-validation described in Section 11.5. However, these two ensembling methods are very different from one another.

With boosting we build up a *single* cross-validated model by gradually *adding* together simple models consisting of a single universal approximator unit (see Section 11.5.4). Each of the constituent models involved in boosting are trained in a way that makes each individual model *dependent* on its predecessors (which are trained first). On the other hand, with bagging (as we have seen) we *average* together *multiple cross-validated* models that have been trained *independently* of each other. Indeed any one of those cross-validated models in a bagged ensemble can itself be a boosted model.

11.10 K-Fold Cross-Validation

In this section we detail a twist on the notion of ensembling, called K-fold cross-validation, that is often applied when human interpretability of a final model is of significant importance. While ensembling often provides a better-fitting averaged predictor that avoids the potential pitfalls of any individual cross-validated model, *human interpretability* is typically lost as the final model is an average of many potentially very different nonlinearities.[14] Instead of *averaging* a set of cross-validated models over many splits of the data, each of which provides minimum validation error over a respective split, with K-fold cross-validation we choose a single model that has minimum *average validation*

[14] Stumps/tree-based approximators are sometimes an exception to this general rule, as detailed in Section 14.2.

error over all splits of the data. This produces a potentially less accurate final model, but one that is significantly simpler (than an ensembled model) and can be more easily understood by humans. As we will see, in special applications K-fold cross-validation is used with *linear* models as well.

11.10.1 The K-folds cross-validation procedure

K-fold cross-validation is a method for determining robust cross-validated models via an ensembling-like procedure that constrains the complexity of the final model so that it is more human interpretable. Instead of averaging a group of cross-validated models, each of which achieves a minimum validation error over a random training–validation split of the data, with K-fold cross-validation we choose a single final model that achieves the *lowest* average validation error over all of the splits together. By selecting a *single* model to represent the entire dataset, as opposed to an *average* of different models (as is done with ensembling), we make it easier to interpret the selected model.

Of course the desire for any nonlinear model to be interpretable means that its fundamental building blocks (universal approximators of a certain type) need to be interpretable as well. Neural networks, for example, are almost never human interpretable while fixed-shape (most commonly polynomials) and tree-based approximators (commonly stumps) can be interpreted depending on the problem at hand. Thus the latter two types of universal approximators are more commonly employed with the K-fold technique.

To further simplify the final outcome of this procedure, instead of using completely random training–validation splits (as done with ensembling) we split the data randomly into a set of K nonoverlapping pieces. This is depicted visually in Figure 11.49 where the original data is split into $K = 3$ nonoverlapping sets.

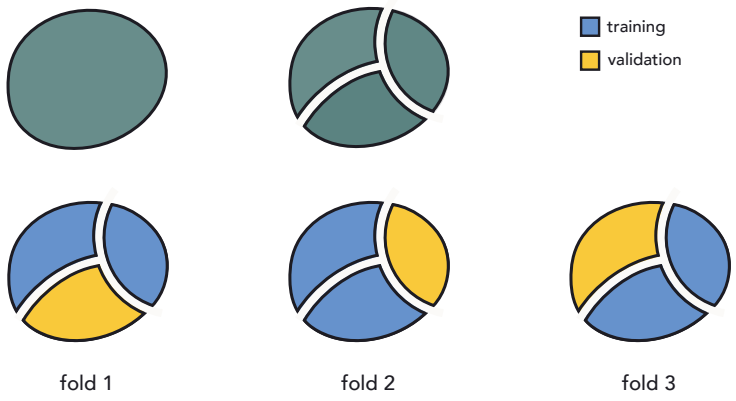

Figure 11.49 Schematic illustration of K-fold cross-validation for $K = 3$.

We then cycle through K training–validation splits of the data that consist of $K - 1$ of these pieces as training, with the final portion as validation, which allows for each point in the dataset to belong to a validation set precisely one time. Each such split is referred to as a *fold*, of which there are K in total, hence the name "K-fold" cross-validation. On each fold we cross-validate the same set of models and record the validation score of each. Afterwards, we choose the single best model that produced the lowest *average validation error*. Once this is done the chosen model is retrained over the entire dataset to provide a final tuned predictor of the data.

Since no models are combined/averaged together with this procedure, it can very easily produce less accurate models (in terms of *testing error* described in Section 11.7) for general learning problems when compared to ensembling. However, when human interpretability of a model overshadows the needs for exceptional performance, K-fold cross-validation produces a stronger-performing model than a single cross-validated model that can still be understood by human beings. This is somewhat analogous to the story of feature selection detailed in Sections 9.6 and 9.7, where human interpretability is the guiding motivator (and not simply prediction power).

Example 11.17 Galileo's gravity experiment

In this example we use K-fold cross-validation on the Galileo dataset detailed in Example 10.2 to recover the quadratic rule that was both engineered there, and that Galileo himself divined from a similar dataset. Since there are only $P = 6$ points in this dataset, intuition suggests that we use a large value for K as described in Section 11.3.4. In this instance we can set K as high as possible, i.e., $K = P$, meaning that each fold will contain only a single data point for validation purposes. This setting of K-fold cross-validation – sometimes referred to as *leave-one-out* cross-validation – is usually employed when the size of data is extremely small.

Here we search over polynomial models of degree 1 through 6, since they are not only easily interpretable, but are appropriate for data gleaned from physical experiments (which often trace out smooth rules). As shown in Figure 11.50, while not all of the models over the six folds fit the data well, the model chosen by K-fold is indeed the quadratic polynomial fit originally proposed by Galileo.

11.10.2 K-fold cross-validation and high-dimensional linear modeling

Suppose for a moment we have a high-capacity model (e.g., a polynomial of degree D where D is very large) which enables several kinds of overfitting behavior for a nonlinear regression dataset, with each overfitting instance of the model provided by different settings of the linear combination weights of the model. We illustrate such a scenario in the left panel of Figure 11.51, where two

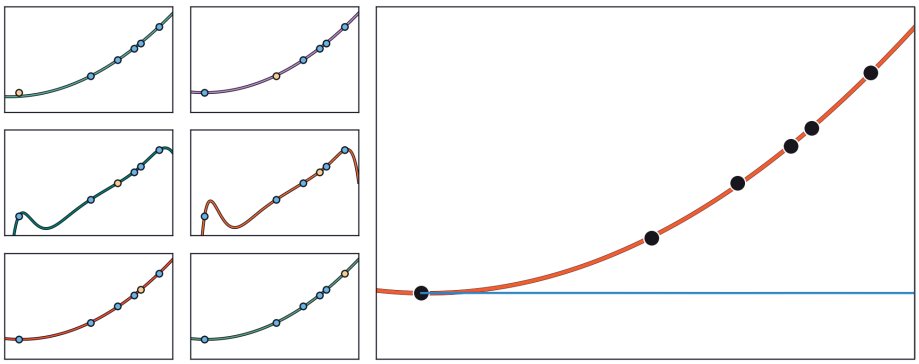

Figure 11.50 Figure associated with Example 11.17. (small panels) Six cross-validated models, each trained on all but one point from the dataset. Here the validation portion of each fold (i.e., a single data point) is highlighted in yellow. (large panel) The model with lowest average validation error is a quadratic. See text for further details.

settings of such a model provide two distinct overfitting predictors for a generic nonlinear regression dataset. As we learned in Section 10.2, any[15] *nonlinear* model in the original space of a regression dataset corresponds to a *linear* model in the transformed feature space (i.e., the space where each individual input axis is given by one of the chosen nonlinear features). Since our model easily overfits the original data, in the transformed feature space our data lies along a *linear subspace* that can be perfectly fit using many different hyperplanes. Indeed the two nonlinear overfitting models shown in the left panel of the figure correspond one-to-one with the two linear fits in the transformed feature space – illustrated symbolically[16] in the right panel of the figure.

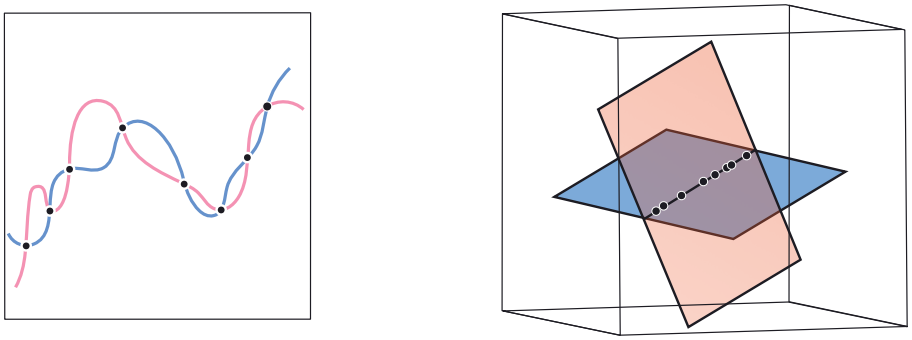

Figure 11.51 (left panel) Two instances of overfitting by a high-capacity model to a nonlinear regression dataset. (right panel) These two models – as viewed in the *feature transformed space* – are linear. See text for further details.

[15] Suppose any parameters internal to the features (if they exist) are fixed.
[16] In reality we could not visualize this space, as it would be too high-dimensional.

The general scenario depicted in the right panel of Figure 11.51 is precisely where we begin when faced with small datasets that have very high input dimension: in such scenarios even a linear model has extremely high capacity and can easily overfit, virtually ruling out the use of more complicated nonlinear models. Thus in such scenarios, in order to properly tune the parameters of a (high-capacity) linear model we often turn to *regularization* to block capacity in high-capacity models (as described in Section 11.6.3). Given the small amount of data at play to determine the best setting of the regularization parameter, K-fold cross-validation is commonly employed to determine the proper regularization parameter value and ultimately the parameters of the linear model.

This scenario provides an interesting point of intersection with the notion of *feature selection via regularization* detailed in Section 9.7. Employing the ℓ_1 regularizer we can block the capacity of our high-capacity linear model while *simultaneously* selecting important input features, facilitating human interpretability of the learned model.

Example 11.18 Genome-wide association studies

Genome-wide association studies (GWAS) aim at understanding the connections between tens of thousands of genetic markers (input features), taken from across the human genome of several subjects, with medical conditions such as high blood pressure, high cholesterol, heart disease, diabetes, various forms of cancer, and many others (see Figure 11.52). These studies typically involve a relatively small number of patients with a given affliction (as compared to the very large dimension of the input). As a result, regularization based cross-validation is a useful tool for learning meaningful (linear) models for such data. Moreover, using a (sparsity-inducing) regularizer like the ℓ_1 norm can help researchers identify the handful of genes critical to the affliction under study, which can both improve our understanding of it and perhaps provoke development of gene-targeted therapies. See Exercise 11.10 for further details.

Example 11.19 fMRI studies

Neuroscientists believe that only a small number of active brain regions are involved in performing any given cognitive task. Therefore limiting the number of input features allowed in the classification model, via ℓ_1 *regularized feature selection*, is commonly done in order to produce high-performing and human-interpretable results. Figure 1.12 illustrates the result of applying a classification model with sparse feature selection to the problem of diagnosing patients with ADHD. The sparsely distributed regions of color represent activation areas uncovered by the learning algorithm that significantly distinguish between individuals with and without ADHD.

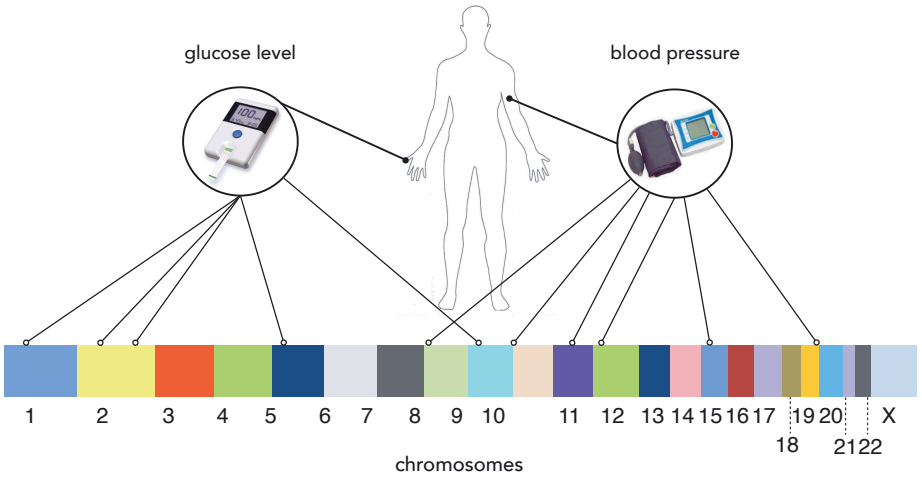

Figure 11.52 Conceptual illustration of a genome-wide association study wherein a quantitative biological trait (e.g., blood pressure or glucose level) is to be associated with specific genomic locations.

11.11 When Feature Learning Fails

Supposing we implement the tools outlined in this chapter correctly, when does cross-validation, ensembling, and (more broadly) feature learning fail? The simple answer is, feature learning fails when our data fails to sufficiently reflect the underlying phenomenon that generated it. Nonlinear feature engineering (outlined in Chapter 10) also fails in such circumstances. This can happen when one or more of the following occur.

- **When a dataset has no inherent structure:** if there is little or no relationship present in the data (due to improper measurement, experimentation, or selection of inputs) the nonlinear model learned via feature learning will be useless. For example, in the left panel of Figure 11.53 we show a small two-class dataset formed by randomly choosing points on the unit square and randomly assigning each point one of two class labels. No classification boundary learned from this dataset can ever yield value, as the data itself contains no meaningful pattern.

- **When a dataset is too small:** when a dataset is too small to represent the true underlying phenomenon feature learning can inadvertently determine an incorrect nonlinearity. For example, in the middle panel of Figure 11.53 we show a simple example of this occurrence. The phenomenon underlying this two-class dataset has a nonlinear boundary (shown in dashed black). However, because we have sampled too few points, the data we do have is linearly

separable and cross-validation will recover a linear boundary (shown in solid black) that does not reflect the true nature of the underlying phenomenon. Because of the small data size this problem is unavoidable.

- **When a dataset is poorly distributed:** even if a dataset is large it can still fail to reflect the true nature of the underlying phenomenon that generated it. When this happens feature learning will fail. For example, in the right panel of Figure 11.53 we show a simple two-class dataset whose two classes are separated by a perfectly circular boundary shown in dashed black. While the dataset is relatively large, the data samples have all been taken from the top portion of the input space. Viewed on its own this reasonably large dataset does not represent the true underlying phenomenon very well. While cross-validation produces a model that perfectly separates the two classes, the corresponding parabolic decision boundary (shown in solid black) does not match the true circular boundary. Such a circumstance is bound to happen when data is poorly distributed and fails to reflect the phenomenon that generated it.

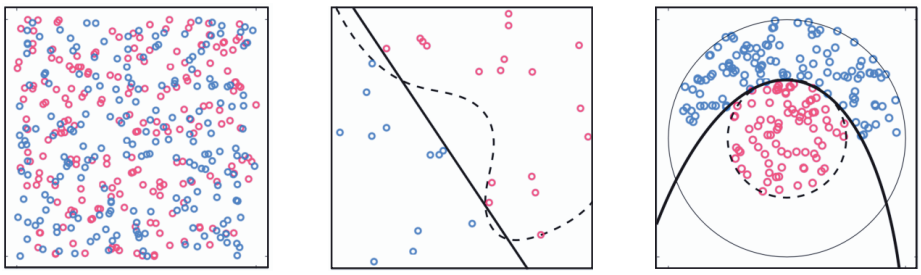

Figure 11.53 Feature learning fails when our data fails to sufficiently reflect the underlying phenomenon that generated it. This can occur when a dataset is poorly structured, too small, or poorly distributed like the data shown in the left, middle, and right panels, respectively. See text for further details.

11.12 Conclusion

This chapter outlined a range of fundamental and extremely important concepts for *feature learning*, or automatic feature engineering, that will echo repeatedly throughout the remainder of the text.

Foremost among these principles is the notion of *universal approximators* introduced in Section 11.2, which are the analog of spanning sets (from vector algebra) for the case of *perfect data*. Here we learned about basic instances of three fundamental families of universal approximators – *fixed-shape*, *neural networks*, and *trees* – each of which has unique attributes and technical eccentricities that are explored in great detail in the chapters following this one. Unlike the case of perfect data, when dealing with *real data* great care must be taken in properly

setting the *capacity* of a model built with units of a universal approximator, and in *optimizing* its parameters appropriately (see Section 11.3). These two "dials" (as they are described in Section 11.3.2) constitute the two main controls we have when properly applying any universal approximator based model to real data. Indeed feature learning is in essence the appropriate setting of the capacity and optimization dials (automatically) via the methods of *cross-validation* detailed in Sections 11.4–11.6. Here we saw that it is easier to fix *capacity* at a high level and carefully *optimize* than vice versa, leading to the *boosting* and *regularization* procedures outlined in Sections 11.5 and 11.6, respectively. Finally, *bagging* – the careful combination of a collection of trained/cross-validated models – was described in Section 11.9 (along with the analogous K-fold cross-validation scheme for smaller, higher-dimensional datasets in Section 11.10), which generally leads to better-preforming (bagged) models.

11.13 Exercises

† The data required to complete the following exercises can be downloaded from the text's github repository at `github.com/jermwatt/machine_learning_refined`

11.1 Naive cross-validation I

Repeat the experiment described in Example 11.8, splitting the original dataset at random into training and validation portions. You need not reproduce the panels in Figure 11.27, but make a plot showing the training and validation errors for your range of models tested, and visualize the model you find (along with the data) that provides the lowest validation error. Given your particular training–validation split your results may be different than those presented in the example.

11.2 Naive cross-validation II

Repeat the experiment described in Example 11.9, splitting the original dataset at random into training and validation portions. You need not re-produce the panels in Figure 11.28, but make a plot showing the training and validation errors for your range of models tested. Given your particular training–validation split your results may be different than those presented in the example.

11.3 Boosting based cross-validation I

Repeat the experiment described in Example 11.11. You need not reproduce the panels in Figure 11.32, but make a plot showing the training and validation errors for your range of models tested.

11.4 Boosting based cross-validation II

Perform 20 rounds of boosting based cross-validation using neural network units (defined in Equation (11.12)), employing the breast cancer dataset discussed in Exercise 9.5, and randomly splitting the original dataset into 80 percent training and 20 percent validation.

11.5 Regularization based cross-validation

Repeat the experiment described in Example 11.13. You need not reproduce the panels in Figure 11.40, but make a plot showing the training and validation errors for your range of models tested.

11.6 Bagging regression models

Repeat the first experiment outlined in Example 11.14, producing ten naively cross-validated polynomial models to fit different training–validation splits of the regression dataset shown in Figure 11.44. Produce a set of plots like the ones shown in Figure 11.44 that show how each individual model fits to the data, as well as how the bagged median model fits.

11.7 Bagging two-class classification models

Repeat the first experiment outlined in Example 11.15, producing five naively cross-validated polynomial models to fit different training–validation splits of the two-class classification dataset shown in Figure 11.46. Compare the efficacy – in terms of number of misclassifications over the entire dataset – of each individual model and the final bagged model.

11.8 Bagging multi-class classification models

Repeat the second experiment outlined in Example 11.16, whose results are shown in the bottom row of Figure 11.48. Compare the efficacy – in terms of number of misclassifications over the entire dataset – of each individual model and the final bagged model.

11.9 K-fold cross-validation

Repeat the experiment outlined in Example 11.17, reproducing the plots shown in Figure 11.50.

11.10 Classification of diabetes

Perform K-fold cross-validation using a linear model and the ℓ_1 regularizer over a popular two-class classification genomics dataset consisting of $P = 72$ datapoints, each of which has input dimension $N = 7128$. This will tend to produce a sparse predictive linear model – as detailed in Example 11.18 – which is helpful

in determining a small number of genes that correlate with the output of this two-class classification dataset (which is whether each individual represented in the dataset has diabetes or not).